Hibernate 中文文档 3.2

wizardforcel

Published with GitBook



目錄

Hibernate 中文文档 3.2

前言

目录

WARNING! This is a translated version of the English Hibernate reference documentation. The translated version might not be up to date! However, the differences should only be very minor. Consult the English reference documentation if you are missing information or encounter a translation error. If you like to contribute to a particular translation, contact us on the Hibernate developer mailing list.

Translator(s): RedSaga Translate Team 满江红翻译团队 <caoxg@yahoo.com>

在今日的企业环境中，把面向对象的软件和关系数据库一起使用可能是相当麻烦、浪费时间的。Hibernate是一个面向Java环境的对象/关系数据库映射工具。对象/关系数据库映射(object/relational mapping (ORM))这个术语表示一种技术，用来把对象模型表示的对象映射到基于SQL的关系模型数据结构中去。

Hibernate不仅仅管理Java类到数据库表的映射（包括Java数据类型到SQL数据类型的映射），还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用SQL和JDBC处理数据的时间。

Hibernate的目标是对于开发者通常的数据持久化相关的编程任务，解放其中的95%。对于以数据为中心的程序来说,它们往往只在数据库中使用存储过程来实现商业逻辑,Hibernate可能不是最好的解决方案;对于那些在基于Java的中间层应用中，它们实现面向对象的业务模型和商业逻辑的应用，Hibernate是最有用的。不管怎样，Hibernate一定可以帮助你消除或者包装那些针对特定厂商的SQL代码，并且帮你把结果集从表格式的表示形式转换到一系列的对象去。

如果你对Hibernate和对象/关系数据库映射还是个新手，或者甚至对Java也不熟悉，请按照下面的步骤来学习。

1. 阅读第 1 章 *Hibernate入门*，这是一篇包含详细的逐步指导的指南。本指南的源代码包含在发行包中，你可以在 `doc/reference/tutorial/` 目录下找到。
2. 阅读第 2 章 体系结构*(Architecture)*来理解Hibernate可以使用的环境。
3. 查看Hibernate发行包中的 `eg/` 目录，里面有一个简单的独立运行的程序。把你的JDBC驱动拷贝到 `lib/` 目录下，修改一下 `src/hibernate.properties` ,指定其中你的数据库的信息。进入命令行，切换到你的发行包的目录，输入 `ant eg` (使用了Ant），或者在Windows操作系统中使用 `build eg` 。
4. 把这份参考文档作为你学习的主要信息来源。
5. 在Hibernate 的网站上可以找到经常提问的问题与解答(FAQ)。

6. 在Hibernate网站上还有第三方的演示、示例和教程的链接。

7. Hibernate网站的“社区(Community Area)”是讨论关于设计模式以及很多整合方案(Tomcat, JBoss AS, Struts, EJB,等等)的好地方。

如果你有问题，请使用Hibernate网站上链接的用户论坛。我们也提供一个JIRA问题追踪系统，来搜集bug报告和新功能请求。如果你对开发Hibernate有兴趣，请加入开发者的邮件列表。（Hibernate网站上的用户论坛有一个中文版面，JavaEye也有Hibernate中文版面,您可以在那里交流问题与经验。）

商业开发、产品支持和Hibernate培训可以通过JBoss Inc.获得。（请查阅：http://www.hibernate.org/SupportTraining/）。 Hibernate是一个专业的开放源代码项目(Professional Open Source project)，也是JBoss Enterprise Middleware System(JEMS),JBoss企业级中间件系统的一个核心组件。

1. 翻译说明

本文档的翻译是在网络上协作进行的，也会不断根据Hibernate的升级进行更新。提供此文档的目的是为了减缓学习Hibernate的坡度，而非代替原文档。我们建议所有有能力的读者都直接阅读英文原文。若您对翻译有异议，或发现翻译错误，敬请不吝赐教，报告到如下email地址：cao at redsaga.com

Hibernate版本3的翻译由满江红翻译团队(RedSaga Translate Team)集体进行，这也是一次大规模网络翻译的试验。在不到20天的时间内，我们完成了两百多页文档的翻译，这一成果是通过十几位网友集体努力完成的。通过这次翻译，我们也有了一套完整的流程，从初译、技术审核一直到文字审核、发布。我们的翻译团队还会继续完善我们的翻译流程，并翻译其他优秀的Java开源资料，敬请期待。

表 1. Hibernate v3翻译团队

序号	标题	中文标题	v3翻译	v3审校	v3.1
--	Quickstart with Tomcat	在Tomcat中快速上手(3.1版本中取消)	曹晓钢	zoujm	--
#1	Turtotial	Hibernate入门	Zheng Shuai	-	Sean
#2	Architecture	体系结构	Hilton(BJUG)	厌倦发呆	Sean
#3	Configuration	配置	Goncha	mochow	zcgly
#4	Persistent Classes	持久化类	曹晓钢	mochow	Digita
#5	Basic O/R Mapping	对象/关系数据库映射基础(上)	moxie	Kingfish	张成钅
		对象/关系数据库映射基础(下)	inter_dudu	刘国雄(vincent)	张成钅
#6	Collection Mapping	集合类映射	曹晓钢	robbin	--
#7	Association Mappings	关联关系映射	Robbin	devils.advocate	--

#8	Component Mapping	组件映射	曹晓钢	Robbin	Song qiang
#9	Inheritance Mappings	继承映射	morning(BJUG)	mochow	Liang cheng
#10	Working with objects	与对象共事	程广楠	厌倦发呆	--
#11	Transactions And Concurrency	事务和并发	Robbin	mochow	--
#12	Interceptors and events	继承映射	七彩狼(BJUG)	厌倦发呆	--
#13	Batch processing	批量处理	Kingfish(BJUG)	厌倦发呆	--
#14	HQL: The Hibernate Query Language	HQL: Hibernate 查询语言	郑浩(BJUG)	Zheng Shuai	--
#15	Criteria Queries	条件查询	nemo(BJUG)	Zheng Shuai	--
#16	Native SQL	Native SQL查询	似水流年	zoujm	--
#17	Filters	过滤数据	冰云(BJUG)	Goncha	--
#18	XML Mapping	XML映射	edward(BJUG)	Goncha	huxb
#19	Improving performance	性能提升	Wangjinfeng	Robbin	--
#20	Toolset Guide	工具箱指南	曹晓钢	Robbin	--
#21	Example: Parent/Child	示例：父子关系	曹晓钢	devils.advocate	--
#22	Example: Weblog Application	示例：Weblog 应用程序	曹晓钢	devils.advocate	--
#23	Example: Various Mappings	示例：多种映射	shidu(BJUG)	冰云	--
#24	Best Practices	最佳实践	曹晓钢	冰云	--

v3.2版本在2006年11月份由曹晓钢更新。

关于我们

满江红.开源, http://www.redsaga.com

从成立之初就致力于Java开放源代码在中国的传播与发展,与国内多个Java团体及出版社有深入交流。坚持少说多做的原则，目前有两个团队，“OpenDoc团队”与“翻译团队”，本翻译文档即为翻译团队作品。OpenDoc团队已经推出包括Hibernate、iBatis、Spring、WebWork的多份开放文档，并于2005年5月在Hibernate开放文档基础上扩充成书，出版了原创书籍：《深入浅出Hibernate》，本书400余页，适合各个层次的Hibernate用户。(http://www.redsaga.com/hibernate_book.html)敬请支持。

北京Java用户组, http://www.bjug.org

Beiing Java User Group，民间技术交流组织，成立于2004年6月。以交流与共享为宗旨，每两周举行一次技术聚会活动。BJUG的目标是，通过小部分人的努力，形成一个技术社群，创建良好的交流氛围，并将新的技术和思想推广到整个IT界，让我们共同进步。

Java视线, http://www.javaeye.com

Java视线在是Hibernate中文论坛（ http://www.hibernate.org.cn ，Hibernate中文论坛是中国最早的Hibernate专业用户论坛，为Hibernate在中国的推广做出了巨大的贡献）基础上发展起来的Java深度技术网站，目标是成为一个高品质的，有思想深度的、原创精神的Java技术交流网站，为软件从业人员提供一个自由的交流技术，交流思想和交流信息的平台。

致谢

还有一些朋友给我们发来了勘误，在此致谢：Kurapica，李毅，李海林。

2. 版权声明

Hibernate英文文档属于Hibernate发行包的一部分，遵循LGPL协议。本翻译版本同样遵循LGPL协议。参与翻译的译者一致同意放弃除署名权外对本翻译版本的其它权利要求。

您可以自由链接、下载、传播此文档，或者放置在您的网站上，甚至作为产品的一部分发行。但前提是必须保证全文完整转载，包括完整的版权信息和作译者声明，并不能违反LGPL协议。这里“完整”的含义是，不能进行任何删除/增添/注解。若有删除/增添/注解，必须逐段明确声明那些部分并非本文档的一部分。

第 1 章 Hibernate入门

目录

1.1. 前言

本章是面向Hibernate初学者的一个入门教程。我们从一个使用驻留内存式(in-memory)数据库的简单命令行应用程序开始, 用易于理解的方式逐步开发。

本章面向Hibernate初学者，但需要Java和SQL知识。它是在Michael Goegl所写的指南的基础上完成的。在这里，我们称第三方库文件是指JDK 1.4和5.0。若使用JDK1.3，你可能需要其它的库文件。

本章的源代码已包含在发布包中，位于 `doc/reference/tutorial/` 目录下。

1.2. 第一部分 － 第一个Hibernate应用程序

首先我们将创建一个简单的基于控制台的(console-based)Hibernate应用程序。由于我们使用Java数据库(HSQL DB)，所以不必安装任何数据库服务器。

假设我们希望有一个小应用程序可以保存我们希望参加的活动（events）和这些活动主办方的相关信息。（译者注：在本教程的后面部分，我们将直接使用event而不是它的中文翻译“活动”，以免混淆。）

我们所做的第一件事就是创建我们的开发目录，并且把所有需要用到的Java库文件放进去。解压缩从Hibernate网站下载的Hibernate发布包，并把 `/lib` 目录下所有需要的库文件拷到我们新建开发目录下的 `/lib` 目录下。看起来就像这样：

```
.
+lib
  antlr.jar
  cglib.jar
  asm.jar
  asm-attrs.jars
  commons-collections.jar
  commons-logging.jar
  ehcache.jar
  hibernate3.jar
  jta.jar
  dom4j.jar
  log4j.jar
```

到编写本文时为止，这些是Hibernate运行所需要的最小库文件集合（注意我们也拷贝了 Hibernate3.jar，这个是最主要的文件）。你正使用的Hibernate版本可能需要比这更多或少一些的库文件。请参见发布包中的 `lib/` 目录下的 `README.txt`，以获取更多关于所需和可选的第三方库文件信息（事实上，Log4j并不是必须的库文件，但被许多开发者所喜欢）。

接下来我们创建一个类，用来代表那些我们希望储存在数据库里的event。

1.2.1. 第一个class

我们的第一个持久化类是一个带有一些属性（property）的简单JavaBean类：

```
package events;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

你可以看到这个类对属性的存取方法（getter and setter method）使用了标准JavaBean命名约定，同时把类属性（field）的访问级别设成私有的（private）。这是推荐的设计，但并不是必须的。Hibernate也可以直接访问这些field，而使用访问方法（accessor method）的好处是提供了重构时的健壮性（robustness）。为了通过反射机制（Reflection）来实例化这个类的对象，我们需要提供一个无参的构造器（no-argument constructor)。

对一特定的event, `id` 属性持有唯一的标识符（identifier）的值。如果我们希望使用Hibernate提供的所有特性，那么所有的持久化实体（persistent entity）类（这里也包括一些次要依赖类）都需要一个这样的标识符属性。而事实上，大多数应用程序（特别是web应用程序）都需要通过标识符来区别对象，所以你应该考虑使用标识符属性而不是把它当作一种限制。然而，我们通常不会操作对象的标识（identity），因此它的setter方法的访问级别应该声明private。这样当对象被保存的时候，只有Hibernate可以为它分配标识符值。你可看到Hibernate可以直接访问public，private和protected的访问方法和field。所以选择哪种方式完全取决于你，你可以使你的选择与你的应用程序设计相吻合。

所有的持久化类（persistent classes）都要求有无参的构造器，因为Hibernate必须使用Java反射机制来为你创建对象。构造器（constructor）的访问级别可以是private，然而当生成运行时代理（runtime proxy）的时候则要求使用至少是package 级别的访问控制，这样在没有字节码指令（bytecode instrumentation）的情况下，从持久化类里获取数据会更有效率。

把这个Java源代码文件放到开发目录下的 `src` 目录里，注意包位置要正确。 现在这个目录看起来应该像这样：

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
```

下一步，我们把这个持久化类的信息告诉Hibernate。

1.2.2. 映射文件

Hibernate需要知道怎样去加载（load）和存储（store）持久化类的对象。这正是Hibernate映射文件发挥作用的地方。映射文件告诉Hibernate它，应该访问数据库(database)里面的哪个表（table）及应该使用表里面的哪些字段（column）。

一个映射文件的基本结构看起来像这样：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd

<hibernate-mapping>
[...]
</hibernate-mapping>
```

注意Hibernate的DTD是非常复杂的。你的编辑器或者IDE里使用它来自动完成那些用来映射的XML元素（element）和属性（attribute）。你也可以在文本编辑器里打开DTD－这是最简单的方式来概览所有的元素和attribute，并查看它们的缺省值以及注释。注意Hibernate不会从web加载DTD文件，但它会首先在应用程序的classpath中查找。DTD文件已包括在 `hibernate3.jar` 里，同时也在Hibernate发布包的 `src/` 目录下。

为缩短代码长度，在以后的例子里我们会省略DTD的声明。当然，在实际的应用程序中，DTD声明是必须的。

在 `hibernate-mapping` 标签（tag）之间, 含有一个 `class` 元素。所有的持久化实体类（再次声明，或许接下来会有依赖类，就是那些次要的实体）都需要一个这样的映射，来把类对象映射到SQL数据库里的表。

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

到目前为止，我们告诉了Hibernate怎样把 `Events` 类的对象持久化到数据库的 `EVENTS` 表里，以及怎样从 `EVENTS` 表加载到 `Events` 类的对象。每个实例对应着数据库表中的一行。现在我们将继续讨论有关唯一标识符属性到数据库表的映射。另外，由于我们不关心怎样处理这个标识符，我们就配置由Hibernate的标识符生成策略来产生代理主键字段。

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

`id` 元素是标识符属性的声明，`name="id"` 声明了Java属性的名字 － Hibernate会使用 `getId()` 和 `setId()` 来访问它。 `column` 属性则告诉Hibernate, 我们使用 `EVENTS` 表的哪个字段作为主键。嵌套的 `generator` 元素指定了标识符生成策略，在这里我们指定 `native` ，它根据已配置的数据库（方言）自动选择最佳的标识符生成策略。Hibernate支持由数据库生成，全局唯一性（globally unique）和应用程序指定（或者你自己为任何已有策略所写的扩展）这些策略来生成标识符。

最后我们在映射文件里面包含需要持久化属性的声明。默认情况下，类里面的属性都被视为非持久化的：

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/
        <property name="title"/>
    </class>

</hibernate-mapping>
```

和 `id` 元素一样，`property` 元素的 `name` 属性告诉Hibernate使用哪个getter和setter方法。在此例中，Hibernate会寻找 `getDate()/setDate()` , 以及 `getTitle()/setTitle()` 。

为什么 `date` 属性的映射含有 `column` attribute，而 `title` 却没有？当没有设定 `column` attribute 的时候，Hibernate缺省地使用JavaBean的属性名作为字段名。对于 `title` ，这样工作得很好。然而， `date` 在多数的数据库里，是一个保留关键字，所以我们最好把它映射成一个不同的名字。

另一有趣的事情是 `title` 属性缺少一个 `type` attribute。我们在映射文件里声明并使用的类型，却不是我们期望的那样，是Java数据类型，同时也不是SQL数据库的数据类型。这些类型就是所谓的Hibernate 映射类型（*mapping types*），它们能把Java数据类型转换到SQL数据类型，反之亦然。再次重申，如果在映射文件中没有设置 `type` 属性的话，Hibernate会自己试着去确定正确的转换类型和它的映射类型。在某些情况下这个自动检测机制（在Java 类上使用反射机制）不会产生你所

期待或需要的缺省值。 `date` 属性就是个很好的例子，Hibernate无法知道这个属性（ `java.util.Date` 类型的）应该被映射成：SQL `date` ，或 `timestamp` ，还是 `time` 字段。在此例中，把这个属性映射成 `timestamp` 转换器，这样我们预留了日期和时间的全部信息。

应该把这个映射文件保存为 `Event.hbm.xml` ，且就在 `Event` Java类的源文件目录下。映射文件可随意地命名，但 `hbm.xml` 的后缀已成为Hibernate开发者社区的约定。现在目录结构看起来应该像这样：

```
.
+lib
  <Hibernate and third-party libraries>
+src
 +events
  Event.java
  Event.hbm.xml
```

我们继续进行Hibernate的主要配置。

1.2.3. Hibernate配置

现在我们已经有了一个持久化类和它的映射文件，该是配置Hibernate的时候了。在此之前，我们需要一个数据库。 HSQL DB是种基于Java 的SQL数据库管理系统（DBMS），可以从HSQL DB的网站上下载。实际上，你只需下载的包中的 `hsqldb.jar` 文件，并把这个文件放在开发文件夹的 `lib/` 目录下即可。

在开发的根目录下创建一个 `data` 目录 － 这是HSQL DB存储数据文件的地方。此时在data目录中运行 `java -classpath ../lib/hsqldb.jar org.hsqldb.Server` 就可启动数据库。你可以在log中看到它的启动，及绑定到TCP/IP套结字，这正是我们的应用程序稍后会连接的地方。如果你希望在本例中运行一个全新的数据库，就在窗口中按下 `CTRL + C` 来关闭HSQL数据库，并删除 `data/` 目录下的所有文件，再重新启动HSQL数据库。

Hibernate是你的应用程序里连接数据库的那层，所以它需要连接用的信息。连接（connection）是通过一个也由我们配置的JDBC连接池（connection pool）来完成的。Hibernate的发布包里包含了许多开源的（open source）连接池，但在我们例子中使用Hibernate内置的连接池。注意，如果你希望使用一个产品级(production-quality)的第三方连接池软件，你必须拷贝所需的库文件到你的classpath下，并使用不同的连接池设置。

为了保存Hibernate的配置，我们可以使用一个简单的 `hibernate.properties` 文件，或者一个稍微复杂的 `hibernate.cfg.xml` ，甚至可以完全使用程序来配置Hibernate。多数用户更喜欢使用XML配置文件：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDri
        <property name="connection.url">jdbc:hsqldb:hsql://localhos
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect<

        <!-- Enable Hibernate's automatic session context managemen
        <property name="current_session_context_class">thread</prop

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.N

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

注意这个XML配置使用了一个不同的DTD。在这里，我们配置了Hibernate的 `SessionFactory` 一一个关联于特定数据库全局的工厂（factory）。如果你要使用多个数据库，就要用多个的 `<session-factory>` ，通常把它们放在多个配置文件中（为了更容易启动）。

最开始的4个 `property` 元素包含必要的JDBC连接信息。方言（dialect）的 `property` 元素指明Hibernate 生成的特定SQL变量。你很快会看到，Hibernate对持久化上下文的自动session管理就会派上用场。 打开 `hbm2ddl.auto` 选项将自

动生成数据库模式（schema）－ 直接加入数据库中。当然这个选项也可以被关闭（通过去除这个配置选项）或者通过Ant任务 `SchemaExport` 的帮助来把数据库schema重定向到文件中。最后，在配置中为持久化类加入映射文件。

把这个文件拷贝到源代码目录下面，这样它就位于classpath的根目录的最后。Hibernate在启动时会自动在classpath的根目录查找名为 `hibernate.cfg.xml` 的配置文件。

1.2.4. 用Ant构建

现在我们用Ant来构建应用程序。你必须先安装Ant－可以从Ant 下载页面得到它。怎样安装Ant就不在这里介绍了，请参考Ant 用户手册。当你安装完了Ant，就可以开始创建 `build.xml` 文件，把它直接放在开发目录下面。

一个简单的build文件看起来像这样：

```
<project name="hibernate-tutorial" default="compile">

    <property name="sourcedir" value="${basedir}/src"/>
    <property name="targetdir" value="${basedir}/bin"/>
    <property name="librarydir" value="${basedir}/lib"/>

    <path id="libraries">
        <fileset dir="${librarydir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${targetdir}"/>
        <mkdir dir="${targetdir}"/>
    </target>

    <target name="compile" depends="clean, copy-resources">
      <javac srcdir="${sourcedir}"
             destdir="${targetdir}"
             classpathref="libraries"/>
    </target>

    <target name="copy-resources">
        <copy todir="${targetdir}">
            <fileset dir="${sourcedir}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>

</project>
```

这将告诉Ant把所有在lib目录下以 `.jar` 结尾的文件拷贝到classpath中以供编译之用。它也把所有的非Java源代码文件，例如配置和Hibernate映射文件，拷贝到目标目录。如果你现在运行Ant，会得到以下输出：

```
C:\hibernateTutorial\>ant
Buildfile: build.xml

copy-resources:
     [copy] Copying 2 files to C:\hibernateTutorial\bin

compile:
    [javac] Compiling 1 source file to C:\hibernateTutorial\bin

BUILD SUCCESSFUL
Total time: 1 second
```

1.2.5. 启动和辅助类

是时候来加载和储存一些 `Event` 对象了，但首先我们得编写一些基础的代码以完成设置。我们必须启动Hibernate，此过程包括创建一个全局的 `SessoinFactory` ，并把它储存在应用程序代码容易访问的地方。 `SessionFactory` 可以创建并打开新的 `Session` 。一个 `Session` 代表一个单线程的单元操作， `SessionFactory` 则是个线程安全的全局对象，只需要被实例化一次。

我们将创建一个 `HibernateUtil` 辅助类（helper class）来负责启动Hibernate和更方便地操作 `SessionFactory` 。让我们来看一下它的实现：

```
package util;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildS
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swal
            System.err.println("Initial SessionFactory creation fai
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

这个类不但在它的静态初始化过程（仅当加载这个类的时候被JVM执行一次）中产生全局的 `SessionFactory` ，而且隐藏了它使用了静态singleton的事实。它也可能在应用程序服务器中的JNDI查找 `SessionFactory` 。

如果你在配置文件中给 `SessionFactory` 一个名字，在 `SessionFactory` 创建后，Hibernate会试着把它绑定到JNDI。要完全避免这样的代码，你也可以使用JMX部署，让具有JMX能力的容器来实例化 `HibernateService` 并把它绑定到JNDI。这些高级可选项在后面的章节中会讨论到。

把 `HibernateUtil.java` 放在开发目录的源代码路径下，与放 `events` 的包并列：

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
  +util
    HibernateUtil.java
  hibernate.cfg.xml
+data
build.xml
```

再次编译这个应用程序应该不会有问题。最后我们需要配置一个日志（logging)系统 — Hibernate使用通用日志接口，允许你在Log4j和JDK 1.4 日志之间进行选择。多数开发者更喜欢Log4j：从Hibernate的发布包中（它在 `etc/` 目录下）拷贝 `log4j.properties` 到你的 `src` 目录，与 `hibernate.cfg.xml` .放在一起。看一下配置示例，如果你希望看到更加详细的输出信息，你可以修改配置。默认情况下，只有Hibernate的启动信息才会显示在标准输出上。

示例的基本框架完成了 — 现在我们可以用Hibernate来做些真正的工作。

1.2.6. 加载并存储对象

我们终于可以使用Hibernate来加载和存储对象了，编写一个带有 `main()` 方法的 `EventManager` 类：

```
package events;
import org.hibernate.Session;

import java.util.Date;

import util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {

        Session session = HibernateUtil.getSessionFactory().getCurr

        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

我们创建了个新的 `Event` 对象并把它传递给Hibernate。现在Hibernate负责与SQL打交道，并把 `INSERT` 命令传给数据库。在运行之前，让我们看一下处理 `Session` 和 `Transaction` 的代码。

一个 `Session` 就是个单一的工作单元。我们暂时让事情简单一些，并假设 Hibernate `Session` 和数据库事务是一一对应的。为了让我们的代码从底层的事务系统中脱离出来（此例中是JDBC，但也可能是JTA），我们使用Hibernate `Session` 中的 `Transaction` API。

`sessionFactory.getCurrentSession()` 是干什么的呢？首先，只要你持有 `SessionFactory` （幸亏我们有 `HibernateUtil` ，可以随时获得），大可在任何时候、任何地点调用这个方法。 `getCurrentSession()` 方法总会返回“当前的”工作单元。记得我们在 `hibernate.cfg.xml` 中把这一配置选项调整为"thread"了吗？因此，因此，当前工作单元被绑定到当前执行我们应用程序的Java线程。但是，这并非是完全准确的,你还得考虑工作单元的生命周期范围(scope),它何时开始,又何时结束.

`Session` 在第一次被使用的时候,即第一次调用 `getCurrentSession()` 的时候,其生命周期就开始。然后它被Hibernate绑定到当前线程。当事务结束的时候，不管是提交还是回滚，Hibernate会自动把 `Session` 从当前线程剥离，并且关闭它。假若你再次调用 `getCurrentSession()` ，你会得到一个新的 `Session` ，并且开始一个新的工作单元。这种线程绑定*(thread-bound)*的编程模型（model）是使用Hibernate的最广泛的方式,因为它支持对你的代码灵活分层(事务划分可以和你的数据访问代码分离开来,在本教程的后面部分就会这么做)。

和工作单元的生命周期这个话题相关，Hibernate `Session` 是否被应该用来执行多次数据库操作？上面的例子对每一次操作使用了一个 `Session` ，这完全是巧合，这个例子不是很复杂，无法展示其他方式。Hibernate `Session` 的生命周期可以很灵活，但是你绝不要把你的应用程序设计成为每一次数据库操作都用一个新的Hibernate `Session` 。因此就算下面的例子（它们都很简单）中你可以看到这种用法，记住每次操作一个*session*是一个反模式。在本教程的后面会展示一个真正的(web)程序。

关于事务处理及事务边界界定的详细信息，请参看第 11 章 事务和并发。在上面的例子中，我们也忽略了所有的错误与回滚的处理。

为第一次运行我们的程序，我们得在Ant的build文件中增加一个可以调用得到的target。

```
<target name="run" depends="compile">
    <java fork="true" classname="events.EventManager" classpathref=
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java>
</target>
```

`action` 参数（argument）的值是通过命令行调用这个target的时候设置的：

```
C:\hibernateTutorial\>ant run -Daction=store
```

你应该会看到，编译以后，Hibernate根据你的配置启动，并产生一大堆的输出日志。在日志最后你会看到下面这行：

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID)
```

这是Hibernate执行的 `INSERT` 命令，问号代表JDBC的绑定参数。如果想要看到绑定参数的值或者减少日志的长度，就要调整你在 `log4j.properties` 文件里的设置。

我们想要列出所有已经被存储的events，就要增加一个条件分支选项到main方法中去。

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println("Event: " + theEvent.getTitle() +
                           " Time: " + theEvent.getDate());
    }
}
```

我们也增加一个新的 `listEvents()` 方法:

```
private List listEvents() {

    Session session = HibernateUtil.getSessionFactory().getCurrentS

    session.beginTransaction();

    List result = session.createQuery("from Event").list();

    session.getTransaction().commit();

    return result;
}
```

我们在这里是用一个HQL（Hibernate Query Language－Hibernate查询语言）查询语句来从数据库中加载所有存在的 `Event` 对象。Hibernate会生成适当的SQL，把它发送到数据库，并操作从查询得到数据的 `Event` 对象。当然，你可以使用HQL来创建更加复杂的查询。

现在，根据以下步骤来执行并测试以上各项：

- 运行 `ant run -Daction=store` 来保存一些内容到数据库。当然，先得用hbm2ddl来生成数据库schema。

- 现在把 `hibernate.cfg.xml` 文件中hbm2ddl属性注释掉，这样我们就取消了在启动时用hbm2ddl来生成数据库schema。通常只有在不断重复进行单元测试的时候才需要打开它，但再次运行hbm2ddl会把你保存的一切都删掉（*drop*）—— `create` 配置的真实含义是："在创建SessionFactory的时候，从schema中drop 掉所有的表，再重新创建它们"。

如果你现在使用命令行参数 `-Daction=list` 运行Ant，你会看到那些至今为止我们所储存的events。当然，你也可以多调用几次 `store` 以保存更多的envents。

注意，很多Hibernate新手在这一步会失败，我们不时看到关于*Table not found*错误信息的提问。但是，只要你根据上面描述的步骤来执行，就不会有这个问题，因为hbm2ddl会在第一次运行的时候创建数据库schema，后继的应用程序重起后还能继续使用这个schema。假若你修改了映射，或者修改了数据库schema，你必须把hbm2ddl重新打开一次。

1.3. 第二部分 － 关联映射

我们已经映射了一个持久化实体类到表上。让我们在这个基础上增加一些类之间的关联。首先我们往应用程序里增加人（people）的概念，并存储他们所参与的一个Event列表。（译者注：与Event一样，我们在后面将直接使用person来表示“人”而不是它的中文翻译）

1.3.1. 映射Person类

最初简单的 `Person` 类：

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id

}
```

创建一个名为 `Person.hbm.xml` 的新映射文件（别忘了最上面的DTD引用）：

```
<hibernate-mapping>

    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

最后，把新的映射加入到Hibernate的配置中：

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

现在我们在这两个实体之间创建一个关联。显然，persons可以参与一系列events，而events也有不同的参加者（persons）。我们需要处理的设计问题是关联方向（directionality），阶数（multiplicity）和集合（collection）的行为。

1.3.2. 单向Set-based的关联

我们将向 `Person` 类增加一连串的events。那样，通过调用 `aPerson.getEvents()` ，就可以轻松地导航到特定person所参与的events，而不用去执行一个显式的查询。我们使用Java的集合类（collection）：`Set` ，因为set 不包含重复的元素及与我们无关的排序。

我们需要用set 实现一个单向多值关联。让我们在Java类里为这个关联编码，接着映射它：

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

在映射这个关联之前，先考虑一下此关联的另外一端。很显然，我们可以保持这个关联是单向的。或者，我们可以在 `Event` 里创建另外一个集合，如果希望能够双向地导航，如：`anEvent.getParticipants()` 。从功能的角度来说，这并不是必须的。因为你总可以显式地执行一个查询，以获得某个特定event的所有参与者。这是个在设计时需要做出的选择，完全由你来决定，但此讨论中关于关联的阶数是清楚的：即两端都是“多”值的，我们把它叫做多对多*(many-to-many)*关联。因而，我们使用Hibernate的多对多映射：

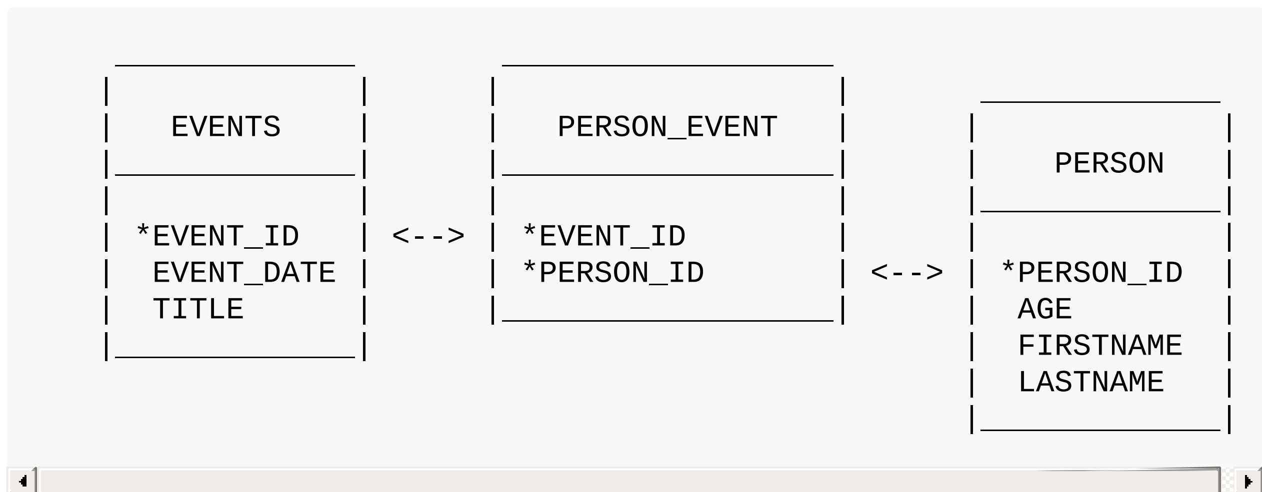
```
<class name="events.Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="events.Event"/>
    </set>

</class>
```

Hibernate支持各种各样的集合映射， <set> 使用的最为普遍。对于多对多关联（或叫*n:m*实体关系）, 需要一个关联表（association table）。 表 里面的每一行代表从person到event的一个关联。表名是由 set 元素的 table 属性配置的。关联里面的标识符字段名，对于person的一端，是由 <key> 元素定义，而event一端的字段名是由 <many-to-many> 元素的 column 属性定义。你也必须告诉Hibernate集合中对象的类（也就是位于这个集合所代表的关联另外一端的类）。

因而这个映射的数据库schema是：



1.3.3. 使关联工作

我们把一些people和events 一起放到 `EventManager` 的新方法中：

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentS
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);

    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

在加载一 `Person` 和 `Event` 后，使用普通的集合方法就可容易地修改我们定义的集合。如你所见，没有显式的 `update()` 或 `save()`，Hibernate会自动检测到集合已经被修改并需要更新回数据库。这叫做自动脏检查（*automatic dirty checking*），你也可以尝试修改任何对象的name或者date属性，只要他们处于持久化状态，也就是被绑定到某个Hibernate 的 `Session` 上（如：他们刚刚在一个单元操作被加载或者保存），Hibernate监视任何改变并在后台隐式写的方式执行SQL。同步内存状态和数据库的过程，通常只在单元操作结束的时候发生，称此过程为清理缓存（*flushing*）。在我们的代码中，工作单元由数据库事务的提交（或者回滚）来结束——这是由 `CurrentSessionContext` 类的 `thread` 配置选项定义的。

当然，你也可以在不同的单元操作里面加载person和event。或在 `Session` 以外修改不是处在持久化（persistent）状态下的对象（如果该对象以前曾经被持久化，那么我们称这个状态为脱管（*detached*））。你甚至可以在一个集合被脱管时修改它：

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentS
    session.beginTransaction();

    Person aPerson = (Person) session
            .createQuery("select p from Person p left join fetch p.
            .setParameter("pid", personId)
            .uniqueResult(); // Eager fetch the collection so we ca

    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collectio

    // Begin second unit of work

    Session session2 = HibernateUtil.getSessionFactory().getCurrent
    session2.beginTransaction();

    session2.update(aPerson); // Reattachment of aPerson

    session2.getTransaction().commit();
}
```

对 `update` 的调用使一个脱管对象重新持久化，你可以说它被绑定到一个新的单元操作上，所以在脱管状态下对它所做的任何修改都会被保存到数据库里。这也包括你对这个实体对象的集合所作的任何改动（增加/删除）。

这对我们当前的情形不是很有用，但它是非常重要的概念，你可以把它融入到你自己的应用程序设计中。在 `EventManager` 的main方法中添加一个新的动作，并从命令行运行它来完成我们所做的练习。如果你需要person及event的标识符 — 那就用 `save()` 方法返回它（你可能需要修改前面的一些方法来返回那个标识符）：

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date())
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event " +
}
```

上面是个关于两个同等重要的实体类间关联的例子。像前面所提到的那样，在特定的模型中也存在其它的类和类型，这些类和类型通常是“次要的”。你已看到过其中的一些，像 `int` 或 `String` 。我们称这些类为值类型（*value type*），它们的实

例依赖（*depend*）在某个特定的实体上。这些类型的实例没有它们自己的标识（identity），也不能在实体间被共享（比如，两个person不能引用同一个 `firstname` 对象，即使他们有相同的first name）。当然，值类型并不仅仅在JDK中存在（事实上，在一个Hibernate应用程序中，所有的JDK类都被视为值类型），而且你也可以编写你自己的依赖类，例如 `Address` ， `MonetaryAmount` 。

你也可以设计一个值类型的集合，这在概念上与引用其它实体的集合有很大的不同，但是在Java里面看起来几乎是一样的。

1.3.4. 值类型的集合

我们把一个值类型对象的集合加入 `Person` 实体中。我们希望保存email地址，所以使用 `String` 类型，而且这次的集合类型又是 `Set` ：

```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

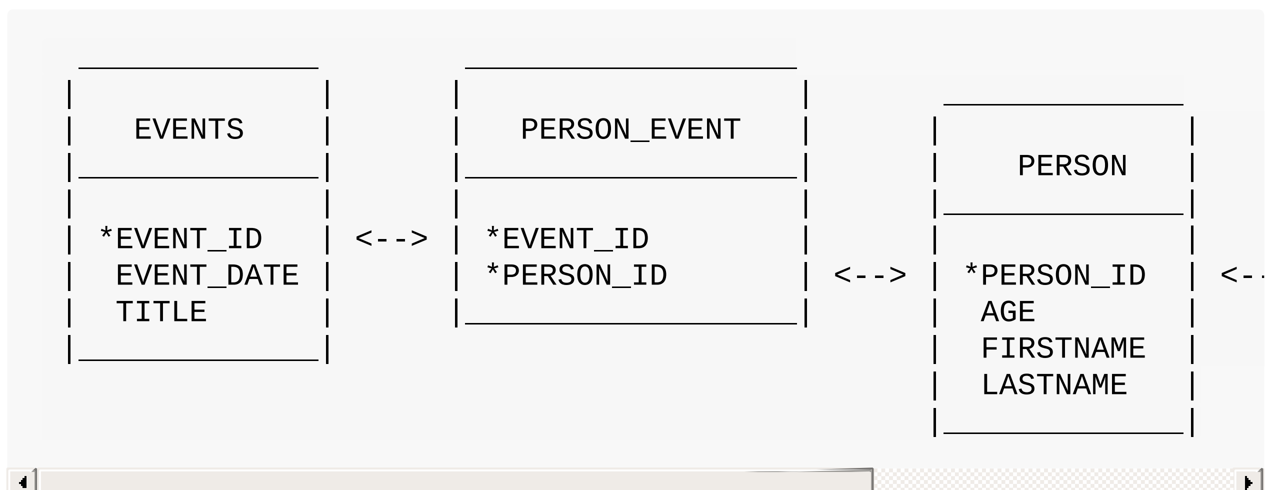
public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

这个 `Set` 的映射

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

比较这次和此前映射的差别，主要在于 `element` 部分，这次并没有包含对其它实体引用的集合，而是元素类型为 `String` 的集合（在映射中使用小写的名字"string"是向你表明它是一个Hibernate的映射类型或者类型转换器）。和之前一样， `set` 元素的 `table` 属性决定了用于集合的表名。 `key` 元素定义了在集合表中外键的字段名。 `element` 元素的 `column` 属性定义用于实际保存 `String` 值的字段名。

看一下修改后的数据库schema。



你可以看到集合表的主键实际上是个复合主键，同时使用了2个字段。这也暗示了对于同一个person不能有重复的email地址，这正是Java里面使用Set时候所需要的语义（Set里元素不能重复）。

你现在可以试着把元素加入到这个集合，就像我们在之前关联person和event的那样。其实现的Java代码是相同的：

```
private void addEmailToPerson(Long personId, String emailAddress) {

    Session session = HibernateUtil.getSessionFactory().getCurrentS
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);

    // The getEmailAddresses() might trigger a lazy load of the col
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

这次我们没有使用*fetch*查询来初始化集合。因此，调用其getter方法会触发另一附加的select来初始化集合，这样我们才能把元素添加进去。检查SQL log，试着通过预先抓取来优化它。

1.3.5. 双向关联

接下来我们将映射双向关联（bi-directional association）－ 在Java里让person和event可以从关联的任何一端访问另一端。当然，数据库schema没有改变，我们仍然需要多对多的阶数。一个关系型数据库要比网络编程语言 更加灵活，所以它并不需要任何像导航方向（navigation direction）的东西 － 数据可以用任何可能的方式进行查看和获取。

首先，把一个参与者（person）的集合加入 `Event` 类中：

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

在 `Event.hbm.xml` 里面也映射这个关联。

```
<set name="participants" table="PERSON_EVENT" inverse="true">
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

如你所见，两个映射文件里都有普通的 `set` 映射。注意在两个映射文件中，互换了 `key` 和 `many-to-many` 的字段名。这里最重要的是 `Event` 映射文件里增加了 `set` 元素的 `inverse="true"` 属性。

这意味着在需要的时候，Hibernate能在关联的另一端 － `Person` 类得到两个实体间关联的信息。这将会极大地帮助你理解双向关联是如何在两个实体间被创建的。

1.3.6. 使双向连起来

首先请记住，Hibernate并不影响通常的Java语义。 在单向关联的例子中，我们是怎样在 `Person` 和 `Event` 之间创建联系的？我们把 `Event` 实例添加到 `Person` 实例内的event引用集合里。因此很显然，如果我们要让这个关联可以双向地工作，我们需要在另外一端做同样的事情 － 把 `Person` 实例加入 `Event` 类内的Person引用集合。这“在关联的两端设置联系”是完全必要的而且你都得这么做。

许多开发人员防御式地编程，创建管理关联的方法来保证正确的设置了关联的两端，比如在 `Person` 里：

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
    event.getParticipants().add(this);
}

public void removeFromEvent(Event event) {
    this.getEvents().remove(event);
    event.getParticipants().remove(this);
}
```

注意现在对于集合的get和set方法的访问级别是protected - 这允许在位于同一个包（package）中的类以及继承自这个类的子类可以访问这些方法，但禁止其他任何人的直接访问，避免了集合内容的混乱。你应尽可能地在另一端也把集合的访问级别设成protected。

`inverse` 映射属性究竟表示什么呢？对于你和Java来说，一个双向关联仅仅是在两端简单地正确设置引用。然而，Hibernate并没有足够的信息去正确地执行 `INSERT` 和 `UPDATE` 语句（以避免违反数据库约束），所以它需要一些帮助来正确的处理双向关联。把关联的一端设置为 `inverse` 将告诉Hibernate忽略关联的这一端，把这端看成是另外一端的一个镜象（*mirror*）。这就是所需的全部信息，Hibernate利用这些信息来处理把一个有向导航模型转移到数据库schema时的所有问题。你只需要记住这个直观的规则：所有的双向关联需要有一端被设置为 `inverse` 。在一对多关联中它必须是代表多（many）的那端。而在多对多（many-to-many）关联中，你可以任意选取一端，因为两端之间并没有差别。

让我们把进入一个小型的web应用程序。

1.4. 第三部分 - EventManager web应用程序

Hibernate web应用程序使用 `Session` 和 `Transaction` 的方式几乎和独立应用程序是一样的。但是，有一些常见的模式（pattern）非常有用。现在我们编写一个 `EventManagerServlet` 。这个servlet可以列出数据库中保存的所有的events，还提供一个HTML表单来增加新的events。

1.4.1. 编写基本的servlet

在你的源代码目录的 `events` 包中创建一个新的类：

```
package events;

// Imports

public class EventManagerServlet extends HttpServlet {

    // Servlet code
}
```

我们后面会用到 `dateFormatter` 的工具，它把 `Date` 对象转换为字符串。只要一个formatter作为servlet的成员就可以了。

这个servlet只处理 HTTP `GET` 请求，因此，我们要实现的是 `doGet()` 方法：

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {

    SimpleDateFormat dateFormatter = new SimpleDateFormat("dd.MM.yy

    try {
        // Begin unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().beginTransaction();

        // Process request and render page...

        // End unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().commit();

    } catch (Exception ex) {
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().rollback();
        throw new ServletException(ex);
    }

}
```

我们称这里应用的模式为每次请求一个session*(session-per-request)*。当有请求到达这个servlet的时候，通过对 `SessionFactory` 的第一次调用，打开一个新的Hibernate `Session` 。然后启动一个数据库事务—所有的数据访问都是在事务中进行，不管是读还是写（我们在应用程序中不使用auto-commit模式）。

不要为每次数据库操作都使用一个新的Hibernate `Session` 。将Hibernate `Session` 的范围设置为整个请求。要用 `getCurrentSession()` ，这样它自动会绑定到当前Java线程。

下一步，对请求的可能动作进行处理，渲染出反馈的HTML。我们很快就会涉及到那部分。

最后，当处理与渲染都结束的时候，这个工作单元就结束了。假若在处理或渲染的时候有任何错误发生，会抛出一个异常，回滚数据库事务。这样， `session-per-request` 模式就完成了。为了避免在每个servlet中都编写事务边界界定的代码，可以考虑写一个servlet 过滤器（filter）来更好地解决。关于这一模式的更多信息，请参阅Hibernate网站和Wiki，这一模式叫做*Open Session in View*—只要你考虑用JSP来渲染你的视图（view），而不是在servlet中，你就会很快用到它。

1.4.2. 处理与渲染

我们来实现处理请求以及渲染页面的工作。

```
// Write HTML header
PrintWriter out = response.getWriter();
out.println("<html><head><title>Event Manager</title></head><body>'

// Handle actions
if ( "store".equals(request.getParameter("action")) ) {

    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b><i>Please enter event title and date.</i></
    } else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(eventDa
        out.println("<b><i>Added event.</i></b>");
    }
}

// Print page
printEventForm(out);
listEvents(out, dateFormatter);

// Write HTML footer
out.println("</body></html>");
out.flush();
out.close();
```

`listEvents()` 方法使用绑定到当前线程的Hibernate `Session` 来执行查询：

```
private void listEvents(PrintWriter out, SimpleDateFormat dateForma

    List result = HibernateUtil.getSessionFactory()
                    .getCurrentSession().createCriteria(Event.class
    if (result.size() > 0) {
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        for (Iterator it = result.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getDate
            out.println("</tr>");
        }
        out.println("</table>");
    }
}
```

最后， `store` 动作会被导向到 `createAndStoreEvent()` 方法，它也使用当前线程的 `Session` :

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
                    .getCurrentSession().save(theEvent);
}
```

大功告成，这个servlet写完了。Hibernate会在单一的 `Session` 和 `Transaction` 中处理到达的servlet请求。如同在前面的独立应用程序中那样，Hibernate可以自动的把这些对象绑定到当前运行的线程中。这给了你用任何你喜欢的方式来对代码分层及访问 `SessionFactory` 的自由。通常，你会用更加完备的设计，把数据访问代码转移到数据访问对象中(DAO模式）。请参见Hibernate Wiki，那里有更多的例子。

1.4.3. 部署与测试

要发布这个程序，你得把它打成web发布包：WAR文件。把下面的脚本加入到你的 `build.xml` 中：

```
<target name="war" depends="compile">
    <war destfile="hibernate-tutorial.war" webxml="web.xml">
        <lib dir="${librarydir}">
          <exclude name="jsdk*.jar"/>
        </lib>

        <classes dir="${targetdir}"/>
    </war>
</target>
```

这段代码在你的开发目录中创建一个 `hibernate-tutorial.war` 的文件。它把所有的类库和 `web.xml` 描述文件都打包进去，web.xml 文件应该位于你的开发根目录中：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>events.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

请注意在你编译和部署web应用程之前，需要一个附加的类库： `jsdk.jar` 。这是Java Servlet开发包，假若你还没有，可以从Sun网站上下载，把它copy到你的lib目录。但是，它仅仅是在编译时需要，不会被打入WAR包。

在你的开发目录中，调用 `ant war` 来构建、打包，然后把 `hibernate-tutorial.war` 文件拷贝到你的tomcat的 `webapps` 目录下。假若你还没安装Tomcat，就去下载一个，按照指南来安装。对此应用的发布，你不需要修改任何Tomcat的配置。

在部署完，启动Tomcat之后，通过 `http://localhost:8080/hibernate-tutorial/eventmanager` 进行访问你的应用，在第一次servlet 请求发生时，请在Tomcat log中确认你看到Hibernate被初始化了（ `HibernateUtil` 的静态初始化器被调用），假若有任何异常抛出，也可以看到详细的输出。

1.5. 总结

本章覆盖了如何编写一个简单独立的Hibernate命令行应用程序及小型的Hibernate web应用程序的基本要素。

如果你已经对Hibernate感到自信，通过开发指南目录，继续浏览你感兴趣的内容－那些会被问到的问题大多是事务处理 (第 11 章 事务和并发)，抓取（fetch）的效率 (第 19 章 提升性能)，或者API的使用 (第 10 章 与对象共事)和查询的特性(第 10.4 节 “查询”)。

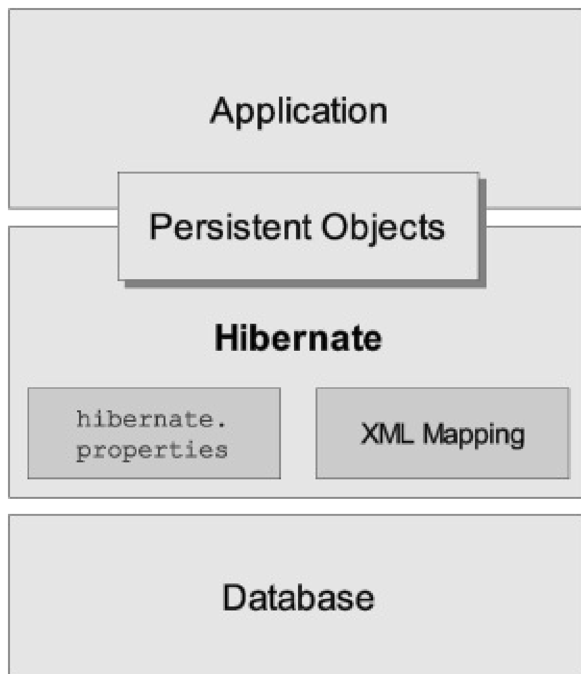
别忘了去Hibernate的网站查看更多（有针对性的）示例。

第 2 章 体系结构(Architecture)

目录

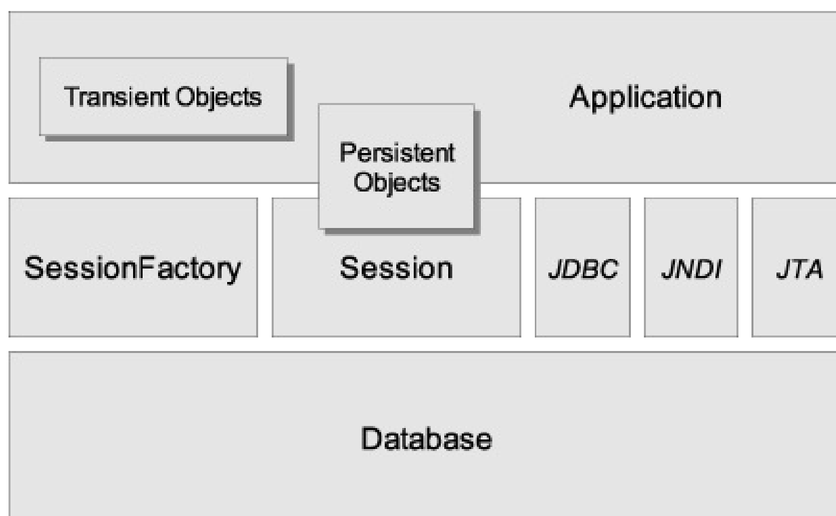
2.1. 概况(Overview)

一个非常简要的Hibernate体系结构的概要图：

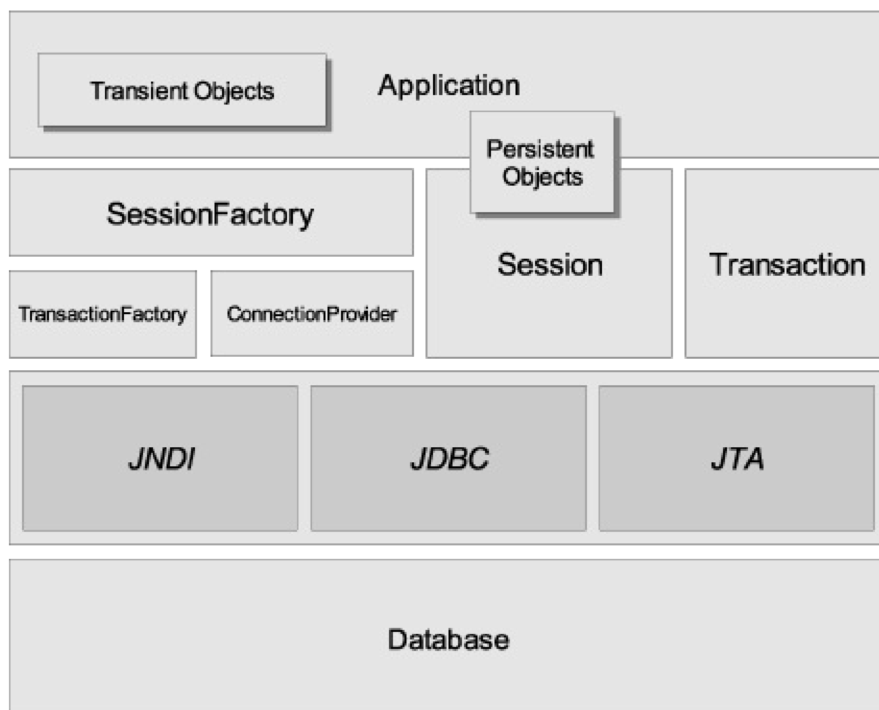


从这个图可以看出，Hibernate使用数据库和配置信息来为应用程序提供持久化服务（以及持久的对象）。

我们来更详细地看一下Hibernate运行时体系结构。由于Hibernate非常灵活，且支持多种应用方案， 所以我们这只描述一下两种极端的情况。“轻型”的体系结构方案，要求应用程序提供自己的JDBC 连接并管理自己的事务。这种方案使用了Hibernate API的最小子集：



“全面解决”的体系结构方案，将应用层从底层的JDBC/JTA API中抽象出来，而让Hibernate来处理这些细节。



图中各个对象的定义如下：

SessionFactory (`org.hibernate.SessionFactory`)

针对单个数据库映射关系经过编译后的内存镜像，是线程安全的（不可变）。 它是生成 `Session` 的工厂，本身要用到 `ConnectionProvider` 。 该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

Session (`org.hibernate.Session`)

表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短。 其隐藏了JDBC连接，也是 `Transaction` 的工厂。 其会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。

持久的对象及其集合

带有持久化状态的、具有业务功能的单线程对象，此对象生存期很短。 这些对象可能是普通的JavaBeans/POJO，唯一特殊的是他们正与（仅仅一个） `Session` 相关联。 一旦这个 `Session` 被关闭，这些对象就会脱离持久化状态，这样就可被应用程序的任何层自由使用。 （例如，用作跟表示层打交道的数据传输对象。）

瞬态(transient)和脱管(detached)的对象及其集合

那些目前没有与session关联的持久化类实例。 他们可能是在被应用程序实例化后，尚未进行持久化的对象。 也可能是因为实例化他们的 `Session` 已经被关闭而脱离持久化的对象。

事务Transaction (`org.hibernate.Transaction`)

（可选的）应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短。 它通过抽象将应用从底层具体的JDBC、JTA以及CORBA事务隔离开。 某些情况下，一个 `Session` 之内可能包含多个 `Transaction` 对象。 尽管是否使用该对象是可选的，但无论是使用底层的API还是使用 `Transaction` 对象，事务边界的开启与关闭是必不可少的。

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)

（可选的）生成JDBC连接的工厂（同时也起到连接池的作用）。 它通过抽象将应用从底层的 `Datasource` 或 `DriverManager` 隔离开。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

TransactionFactory (`org.hibernate.TransactionFactory`)

（可选的）生成 `Transaction` 对象实例的工厂。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

扩展接口

Hibernate提供了很多可选的扩展接口，你可以通过实现它们来定制你的持久层的行为。 具体请参考API文档。

在特定“轻型”的体系结构中，应用程序可能绕过 `Transaction` / `TransactionFactory` 以及 `ConnectionProvider` 等API直接跟JTA或JDBC打交道。

2.2. 实例状态

一个持久化类的实例可能处于三种不同状态中的某一种。 这三种状态的定义则与所谓的持久化上下文*(persistence context)*有关。 Hibernate的 `Session` 对象就是这个所谓的持久化上下文：

瞬态（transient）

该实例从未与任何持久化上下文关联过。它没有持久化标识（相当于主键值）。

持久化(persistent)

实例目前与某个持久化上下文有关联。 它拥有持久化标识（相当于主键值），并且可能在数据库中有一个对应的行。 对于某一个特定的持久化上下文，Hibernate保证持久化标识与Java标识（其值代表对象在内存中的位置）等价。

脱管(detached)

实例曾经与某个持久化上下文发生过关联，不过那个上下文被关闭了， 或者这个实例是被序列化(serialize)到另外的进程。 它拥有持久化标识，并且在数据库中可能存在一个对应的行。 对于脱管状态的实例，Hibernate不保证任何持久化标识和Java标识的关系。

2.3. JMX整合

JMX是管理Java组件(Java components)的J2EE标准。 Hibernate 可以通过一个JMX标准服务来管理。 在这个发行版本中，我们提供了一个MBean接口的实现,即 `org.hibernate.jmx.HibernateService` 。

想要看如何在JBoss应用服务器上将Hibernate部署为一个JMX服务的例子，您可以参考JBoss用户指南。 我们现在说一下在Jboss应用服务器上，使用JMX来部署Hibernate的好处：

- *Session*管理： Hibernate的 `Session` 对象的生命周期可以 自动跟一个JTA事务边界绑定。这意味着你无需手工开关 `Session` 了, 这项 工作会由JBoss EJB 拦截器来完成。你再也不用担心你的代码中的事务边界了(除非你想利用Hibernate提供可选 的 `Transaction` API来自己写一个便于移植的的持久层)。 你通过调用 `HibernateContext` 来访问 `Session` 。
- *HAR* 部署*:* 通常情况下，你会使用JBoss的服务部署描述符（在EAR或/和SAR文件中）来部署Hibernate JMX服务。 这种部署方式支持所有常见的Hibernate `SessionFactory` 的配置选项。 不过，你仍需在部署描述符中，列出你所有的映射文件的名字。如果你使用HAR部署方式, JBoss 会自动探测出你的HAR文件中所有的映射文件。

这些选项更多的描述，请参考JBoss 应用程序用户指南。

将Hibernate以部署为JMX服务的另一个好处，是可以查看Hibernate的运行时统计信息。参看 第 3.4.6 节 “ Hibernate的统计(statistics)机制 ”.

2.4. 对JCA的支持

Hibernate也可以被配置为一个JCA连接器（JCA connector）。更多信息请参看网站。 请注意，Hibernate对JCA的支持，仍处于实验性阶段。

2.5. 上下文相关的（Contextual）Session

使用Hibernate的大多数应用程序需要某种形式的“上下文相关的” session，特定的session在整个特定的上下文范围内始终有效。然而，对不同类型的应用程序而言，要为什么是组成这种“上下文”下一个定义通常是困难的；不同的上下文对“当前”这个概念定义了不同的范围。在3.0版本之前，使用Hibernate的程序要么采用自行编写的基于 `ThreadLocal` 的上下文session，要么采用 `HibernateUtil` 这样的辅助类，要么采用第三方框架（比如Spring或Pico），它们提供了基于代理(proxy)或者基于拦截器(interception)的上下文相关session。

从3.0.1版本开始，Hibernate增加了 `SessionFactory.getCurrentSession()` 方法。一开始，它假定了采用 `JTA` 事务， `JTA` 事务定义了当前session的范围和上下文(scope and context)。Hibernate开发团队坚信，因为有好几个独立的 `JTA TransactionManager` 实现稳定可用，不论是否被部署到一个 `J2EE` 容器中，大多数(假若不是所有的）应用程序都应该采用 `JTA` 事务管理。基于这一点，采用 `JTA` 的上下文相关session可以满足你一切需要。

更好的是，从3.1开始， `SessionFactory.getCurrentSession()` 的后台实现是可拔插的。因此，我们引入了新的扩展接口 (`org.hibernate.context.CurrentSessionContext`)和新的配置参数 (`hibernate.current_session_context_class`)，以便对什么是“当前session”的范围和上下文(scope and context)的定义进行拔插。

请参阅 `org.hibernate.context.CurrentSessionContext` 接口的Javadoc,那里有关于它的契约的详细讨论。它定义了单一的方法， `currentSession()` ，特定的实现用它来负责跟踪当前的上下文session。Hibernate内置了此接口的三种实现。

- `org.hibernate.context.JTASessionContext` - 当前session根据 `JTA` 来跟踪和界定。这和以前的仅支持JTA的方法是完全一样的。详情请参阅Javadoc。
- `org.hibernate.context.ThreadLocalSessionContext` - 当前session通过当前执行的线程来跟踪和界定。详情也请参阅Javadoc。
- `org.hibernate.context.ManagedSessionContext` - 当前session通过当前执行的线程来跟踪和界定。但是，你需要负责使用这个类的静态方法将 `Session` 实例绑定、或者取消绑定，它并不会打开(open)、flush或者关闭(close)任何 `Session` 。

前两种实现都提供了“每数据库事务对应一个session”的编程模型，也称作每次请求一个*session*。Hibernate session的起始和终结由数据库事务的生存来控制。假若你在纯粹的 Java SE之上采用自行编写代码来管理事务,而不使用JTA，建议你使用Hibernate `Transaction` API来把底层事务实现从你的代码中隐藏掉。如果你使用JTA，请使用JTA借口来管理Transaction。如果你在支持CMT的EJB容器中执行代码，事务边界是声明式定义的，你不需要在代码中进行任何事务或session管理操作。请参阅第 11 章 事务和并发一节来阅读更多的内容和示例代码。

`hibernate.current_session_context_class` 配置参数定义了应该采用哪个 `org.hibernate.context.CurrentSessionContext` 实现。注意，为了向下兼容，如果未配置此参数，但是存在 `org.hibernate.transaction.TransactionManagerLookup` 的配置，Hibernate会采用 `org.hibernate.context.JTASessionContext` 。一般而言，此参数的值指明了要使用的实现类的全名，但那三种内置的实现可以使用简写，即"jta"、"thread"和"managed"。

第 3 章 配置

目录

由于Hibernate是为了能在各种不同环境下工作而设计的, 因此存在着大量的配置参数. 幸运的是多数配置参数都 有比较直观的默认值, 并有随Hibernate一同分发的配置样例 `hibernate.properties` (位于 `etc/`)来展示各种配置选项. 所需做的仅仅是将这个样例文件复制到类路径 (classpath)下并做一些自定义的修改.

3.1. 可编程的配置方式

一个 `org.hibernate.cfg.Configuration` 实例代表了一个应用程序中Java类型到SQL数据库映射的完整集合. `Configuration` 被用来构建一个(不可变的(immutable)) `SessionFactory` . 映射定义则由不同的XML映射定义文件编译而来.

你可以直接实例化 `Configuration` 来获取一个实例，并为它指定XML映射定义 文件. 如果映射定 义文件在类路径(classpath)中, 请使用 `addResource()` :

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

一个替代方法（有时是更好的选择）是，指定被映射的类，让Hibernate帮你寻找映射定义文件:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate将会在类路径(classpath)中寻找名字为 `/org/hibernate/auction/Item.hbm.xml` 和 `/org/hibernate/auction/Bid.hbm.xml` 映射定义文件. 这种方式消除了任何对文件名的硬编码(hardcoded).

`Configuration` 也允许你指定配置属性:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLI
    .setProperty("hibernate.connection.datasource", "java:comp/env/
    .setProperty("hibernate.order_updates", "true");
```

当然这不是唯一的传递Hibernate配置属性的方式, 其他可选方式还包括:

1. 传一个 `java.util.Properties` 实例给 `Configuration.setProperties()` .
2. 将 `hibernate.properties` 放置在类路径(classpath)的根目录下 (root directory).
3. 通过 `java -Dproperty=value` 来设置系统 (`System`)属性.

4. 在 hibernate.cfg.xml 中加入元素 <property> (稍后讨论).

如果想尽快体验Hibernate, hibernate.properties 是最简单的方式.

Configuration 实例被设计成启动期间（startup-time）对象, 一旦 SessionFactory 创建完成它就被丢弃了.

3.2. 获得SessionFactory

当所有映射定义被 `Configuration` 解析后, 应用程序必须获得一个用于构造 `Session` 实例的工厂. 这个工厂将被应用程序的所有线程共享:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate允许你的应用程序创建多个 `SessionFactory` 实例. 这对 使用多个数据库的应用来说很有用.

3.3. JDBC连接

通常你希望 `SessionFactory` 来为你创建和缓存(pool)JDBC连接. 如果你采用这种方式, 只需要如下例所示那样，打开一个 `Session` :

```
Session session = sessions.openSession(); // open a new Session
```

一旦你需要进行数据访问时, 就会从连接池(connection pool)获得一个JDBC连接.

为了使这种方式工作起来, 我们需要向Hibernate传递一些JDBC连接的属性. 所有Hibernate属性的名字和语义都在 `org.hibernate.cfg.Environment` 中定义. 我们现在将描述JDBC连接配置中最重要的设置.

如果你设置如下属性，Hibernate将使用 `java.sql.DriverManager` 来获得(和缓存)JDBC连接 :

表 3.1. Hibernate JDBC属性

属性名	用途
`hibernate.connection.driver_class`	*jdbc驱动类*
`hibernate.connection.url`	*jdbc URL*
`hibernate.connection.username`	数据库用户
`hibernate.connection.password`	数据库用户密码
`hibernate.connection.pool_size`	连接池容量上限数目

但Hibernate自带的连接池算法相当不成熟. 它只是为了让你快些上手，并不适合用于产品系统或性能测试中。 出于最佳性能和稳定性考虑你应该使用第三方的连接池。只需要用特定连接池的设置替换 `hibernate.connection.pool_size` 即可。这将关闭Hibernate自带的连接池. 例如, 你可能会想用C3P0.

C3P0是一个随Hibernate一同分发的开源的JDBC连接池, 它位于 `lib` 目录下。 如果你设置了 `hibernate.c3p0.*` 相关的属性, Hibernate将使用 `C3P0ConnectionProvider` 来缓存JDBC连接. 如果你更原意使用Proxool, 请参考发 行包中的 `hibernate.properties` 并到Hibernate网站获取更多的信息.

这是一个使用C3P0的 `hibernate.properties` 样例文件:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

为了能在应用程序服务器(application server)中使用Hibernate, 应当总是将Hibernate 配置成从注册在JNDI中的 `Datasource` 处获得连接，你至少需要设置下列属性中的一个:

表 3.2. Hibernate数据源属性

属性名	用途
`hibernate.connection.datasource`	数据源*JNDI*名字
`hibernate.jndi.url`	*JNDI*提供者的*URL* (可选)
`hibernate.jndi.class`	*JNDI* *`InitialContextFactory`* 类 (可选)
`hibernate.connection.username`	数据库用户 (可选)
`hibernate.connection.password`	数据库用户密码 (可选)

这是一个使用应用程序服务器提供的JNDI数据源的 `hibernate.properties` 样例文件:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

从JNDI数据源获得的JDBC连接将自动参与到应用程序服务器中容器管理的事务(container-managed transactions)中去.

任何连接(connection)属性的属性名都要以" `hibernate.connnection` "开头. 例如, 你可能会使用 `hibernate.connection.charSet` 来指定字符集 `charSet` .

通过实现 `org.hibernate.connection.ConnectionProvider` 接口，你可以定义属于 你自己的获得JDBC连接的插件策略。通过设置 `hibernate.connection.provider_class` ， 你可以选择一个自定义的实现.

3.4. 可选的配置属性

有大量属性能用来控制Hibernate在运行期的行为. 它们都是可选的, 并拥有适当的默认值.

警告: 其中一些属性是"系统级(*system-level*)的". 系统级属性只能通过 `java -Dproperty=value` 或 `hibernate.properties` 来设置, 而不能用上面描述的其他方法来设置.

表 3.3. Hibernate配置属性

属性名	用途
`hibernate.dialect`	一个Hibernate `Dialect` 类名允许Hibernate针对特定的关系数据库生成优化的SQL. 取值 `full.classname.of.Dialect`
`hibernate.show_sql`	输出所有SQL语句到控制台. 有一个另外的选择是把 `org.hibernate.SQL` 这个log category设为 `debug` 。 **eg.** `true` | `false`
`hibernate.format_sql`	在log和console中打印出更漂亮的SQL。 取值 `true` | `false`
`hibernate.default_schema`	在生成的SQL中, 将给定的schema/tablespace附加于非全限定名的表名上. 取值 `SCHEMA_NAME`
`hibernate.default_catalog`	在生成的SQL中, 将给定的catalog附加于非全限定名的表名上. 取值 `CATALOG_NAME`
`hibernate.session_factory_name`	`SessionFactory` 创建后，将自动使用这个名字绑定到JNDI中. 取值 `jndi/composite/name`
`hibernate.max_fetch_depth`	为单向关联(一对一, 多对一)的外连接抓取（outer join fetch）树设置最大深度. 值为 `0` 意味着将关闭默认的外连接抓取. 取值 建议在 `0` 到 `3` 之间取值
`hibernate.default_batch_fetch_size`	为Hibernate关联的批量抓取设置默认数量. 取值 建议的取值

	为 4 , 8 ,和 16
hibernate.default_entity_mode	为由这个 SessionFactory 打开的所有Session指定默认的实体表现模式. 取值 dynamic-map , dom4j , pojo
hibernate.order_updates	强制Hibernate按照被更新数据的主键，为SQL更新排序。这么做将减少在高并发系统中事务的死锁。 取值 true | false
hibernate.generate_statistics	如果开启, Hibernate将收集有助于性能调节的统计数据. 取值 true | false
hibernate.use_identifer_rollback	如果开启, 在对象被删除时生成的标识属性将被重设为默认值. 取值 true | false
hibernate.use_sql_comments	如果开启, Hibernate将在SQL中生成有助于调试的注释信息, 默认值为 false . 取值 true | false

表 3.4. Hibernate JDBC和连接(connection)属性

属性名	用途
hibernate.jdbc.fetch_size	非零值，指定JDBC抓取数量 用 Statement.setFetchSi
hibernate.jdbc.batch_size	非零值，允许Hibernate使用 建议取 5 到 30 之间的值
hibernate.jdbc.batch_versioned_data	如果你想让你的JDBC驱动从 正确的行计数 , 那么将此属性 选项通常是安全的). 同时，H 化的数据使用批量DML. 默认 true | false
hibernate.jdbc.factory_class	选择一个自定义的 Batcher 这个配置属性. **eg.** classna
hibernate.jdbc.use_scrollable_resultset	允许Hibernate使用JDBC2的 用用户提供的JDBC连接时， 则Hibernate会使用连接的元 false
	在JDBC读

hibernate.jdbc.use_streams_for_binary	写 binary (二进制) 或 se 的类型时使用流(stream)(系 false
hibernate.jdbc.use_get_generated_keys	在数据插入数据库之后，允 PreparedStatement.getG 取数据库生成的key(键)。需 JRE1.4+, 如果你的数据库驱 识生成器时遇到问题，请将 下将使用连接的元数据来判 true|false
hibernate.connection.provider_class	自定义 ConnectionProvide Hibernate提供JDBC连接. 取 classname.of.Connectio
hibernate.connection.isolation	设置JDBC事务隔离级别. 查 看 java.sql.Connection 义, 但请注意多数数据库都不 值 1, 2, 4, 8
hibernate.connection.autocommit	允许被缓存的JDBC连接开启 (不建议). 取值 true | fa
hibernate.connection.release_mode	指定Hibernate在何时释放JD 到Session被显式关闭或被断 JDBC连接. 对于应用程序服 当使用 after_statement , 后，都会主动的释放连接. 对 用 after_transaction 在 是合理的. auto 将为JTA和 择 after_statement , 为J 择 after_transaction .

取值 auto (默认) | on_close | after_transaction | after_statement

注意,这些设置仅对通过 SessionFactory.openSession 得到的 Session 起作用。对于通过 SessionFactory.getCurrentSession 得到的 Session ，所配置的 CurrentSessionContext 实现控制这些 Session 的连接释放模式。请参阅第 2.5 节 “上下文相关的（Contextual）Session”。

|| hibernate.connection._<propertyName>_ | 将JDBC属性 propertyName 传递到 DriverManager.getConnection() 中去. || hibernate.jndi._<propertyName>_ | 将属性 propertyName 传递到 JNDI InitialContextFactory 中去. |

表 3.5. Hibernate缓存属性

属性名	用途
`hibernate.cache.provider_class`	自定义的 `CacheProvider` 的名. 取值 `classname.of.CacheProvi`
`hibernate.cache.use_minimal_puts`	以频繁的读操作为代价, 优化: 缓存来最小化写操作. 在 Hibernate3中，这个设置对的 缓存非常有用, 对集群缓存的 而言，默认是开启的. 取值 `true|false`
`hibernate.cache.use_query_cache`	允许查询缓存, 个别查询仍然 被设置为可缓存的. 取值 `true|false`
`hibernate.cache.use_second_level_cache`	能用来完全禁止使用二级缓存 那些在类的映射定义中指 定 `<cache>` 的类，会 认开启二级缓存. 取值 `true|false`
`hibernate.cache.query_cache_factory`	自定义实现 `QueryCache` 接口 类名, 默认为内建 的 `StandardQueryCache` . 取 `classname.of.QueryCache`
`hibernate.cache.region_prefix`	二级缓存区域名的前缀. 取值 `prefix`
`hibernate.cache.use_structured_entries`	强制Hibernate以更人性化的格 将数据存入二级缓存. 取值 `true|false`

表 3.6. Hibernate事务属性

属性名	
`hibernate.transaction.factory_class`	一个 `TransactionFa` Hibernate `Transact` 为 `JDBCTransactio` `classname.of.Tra`
`jta.UserTransaction`	一个JNDI名字，被 `JTATransaction` 器获取JTA `UserTra` `jndi/composite/n`
`hibernate.transaction.manager_lookup_class`	一个 `TransactionMa` 使用JVM级缓存，或 器的时候需要该类. 耳 `classname.of.Tra`
`hibernate.transaction.flush_before_completion`	如果开启, session在 (flush)。 现在更好的 下文管理。请参见第（Contextual）Sess `false`
`hibernate.transaction.auto_close_session`	如果开启, session在 闭。 现在更好的方法 管理。请参见第 2.5（Contextual）Sess `false`

表 **3.7.** 其他属性

属性名	
`hibernate.current_session_context_class`	为"当前" `Session` 指定一 情，请参见第 2.5 节 "上下 `jta` | `thread` | `manag`
`hibernate.query.factory_class`	选择HQL解析器的实现. 取 `org.hibernate.hql.ast` `org.hibernate.hql.cla`
`hibernate.query.substitutions`	将Hibernate查询中的符号日 名或常量名字). 取值 `hqlLiteral=SQL_LITERA`
`hibernate.hbm2ddl.auto`	在 `SessionFactory` 创建 schema的DDL导出到数据 闭 `SessionFactory` 时， | `update` | `create` | `c`
`hibernate.cglib.use_reflection_optimizer`	开启CGLIB来替代运行时反 时比较有用. 注意即使关闭 能在 `hibernate.cfg.xml`

3.4.1. SQL方言

你应当总是为你的数据库将 `hibernate.dialect` 属性设置成正确的 `org.hibernate.dialect.Dialect` 子类. 如果你指定一种方言, Hibernate将为上面列出的一些属性使用合理的默认值, 为你省去了手工指定它们的功夫.

表 **3.8. Hibernate SQL方言 (`hibernate.dialect`)**

RDBMS	方言
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2. 外连接抓取(Outer Join Fetching)

如果你的数据库支持ANSI, Oracle或Sybase风格的外连接, 外连接抓取通常能通过限制往返数据库次数 (更多的工作交由数据库自己来完成)来提高效率. 外连接抓取允许在单个 `SELECT` SQL语句中， 通过many-to-one, one-to-many, many-to-many和one-to-one关联获取连接对象的整个对象图.

将 `hibernate.max_fetch_depth` 设为 `0` 能在全局 范围内禁止外连接抓取. 设为 `1` 或更高值能启用one-to-one和many-to-oneouter关联的外连接抓取, 它们通过 `fetch="join"` 来映射.

参见第 19.1 节 “ 抓取策略(Fetching strategies) ”获得更多信息.

3.4.3. 二进制流 (Binary Streams)

Oracle限制那些通过JDBC驱动传输的 `字节` 数组的数目. 如果你希望使用 `二进值 (binary)` 或 `可序列化的 (serializable)` 类型的大对象, 你应该开启 `hibernate.jdbc.use_streams_for_binary` 属性. 这是系统级属性.

3.4.4. 二级缓存与查询缓存

以 `hibernate.cache` 为前缀的属性允许你在Hibernate中，使用进程或群集范围内的二级缓存系统. 参见第 19.2 节 “二级缓存（The Second Level Cache）”获取更多的详情.

3.4.5. 查询语言中的替换

你可以使用 `hibernate.query.substitutions` 在Hibernate中定义新的查询符号. 例如:

```
hibernate.query.substitutions true=1, false=0
```

将导致符号 `true` 和 `false` 在生成的SQL中被翻译成整数常量.

```
hibernate.query.substitutions toLowercase=LOWER
```

将允许你重命名SQL中的 `LOWER` 函数.

3.4.6. Hibernate的统计(statistics)机制

如果你开启 `hibernate.generate_statistics`，那么当你通过 `SessionFactory.getStatistics()` 调整正在运行的系统时，Hibernate将导出大量有用的数据. Hibernate甚至能被配置成通过JMX导出这些统计信息. 参考 `org.hibernate.stats` 中接口的Javadoc，以获得更多信息.

3.5. 日志

Hibernate使用Apache commons-logging来为各种事件记录日志.

commons-logging将直接输出到Apache Log4j(如果在类路径中包括 `log4j.jar`) 或 JDK1.4 logging (如果运行在JDK1.4或以上的环境下). 你可以从 `http://jakarta.apache.org` 下载Log4j. 要使用Log4j，你需要将 `log4j.properties` 文件放置在类路径下, 随Hibernate 一同分发的样例属性文件在 `src/` 目录下.

我们强烈建议你熟悉一下Hibernate的日志消息. 在不失可读性的前提下， 我们做了很多工作，使Hibernate的日志可能地详细. 这是必要的查错利器. 最令人感兴趣的日志分类有如下这些:

表 3.9. Hibernate日志类别

类别	功能
`org.hibernate.SQL`	在所有SQL DML语句被执行时为它们记录日志
`org.hibernate.type`	为所有JDBC参数记录日志
`org.hibernate.tool.hbm2ddl`	在所有SQL DDL语句执行时为它们记录日志
`org.hibernate.pretty`	在session清洗(flush)时，为所有与其关联的实体(最多20个)的状态记录日志
`org.hibernate.cache`	为所有二级缓存的活动记录日志
`org.hibernate.transaction`	为事务相关的活动记录日志
`org.hibernate.jdbc`	为所有JDBC资源的获取记录日志
`org.hibernate.hql.AST`	在解析查询的时候,记录HQL和SQL的AST分析日志
`org.hibernate.secure`	为JAAS认证请求做日志
`org.hibernate`	为任何Hibernate相关信息做日志 (信息量较大, 但对查错非常有帮助)

在使用Hibernate开发应用程序时, 你应当总是为 `org.hibernate.SQL` 开启 `debug` 级别的日志记录,或者开启 `hibernate.show_sql` 属性。

3.6. 实现 `NamingStrategy`

`org.hibernate.cfg.NamingStrategy` 接口允许你为数据库中的对象和schema元素指定一个“命名标准”.

你可能会提供一些通过Java标识生成数据库标识或将映射定义文件中"逻辑"表/列名处理成"物理"表/列名的规则. 这个特性有助于减少冗长的映射定义文件.

在加入映射定义前，你可以调用 `Configuration.setNamingStrategy()` 指定一个不同的命名策略:

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`org.hibernate.cfg.ImprovedNamingStrategy` 是一个内建的命名策略, 对 一些应用程序而言，可能是非常有用的起点.

3.7. XML配置文件

另一个配置方法是在 `hibernate.cfg.xml` 文件中指定一套完整的配置. 这个文件可以当成 `hibernate.properties` 的替代。 若两个文件同时存在，它将覆盖前者的属性.

XML配置文件被默认是放在 `CLASSPATH` 的根目录下. 这是一个例子:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.d

<hibernate-configuration>

    <!-- 以/jndi/name绑定到JNDI的SessionFactory实例 -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- 属性 -->
        <property name="connection.datasource">java:/comp/env/jdbc/
        <property name="dialect">org.hibernate.dialect.MySQLDialect
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransact

        <!-- 映射定义文件 -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- 缓存设置 -->
        <class-cache class="org.hibernate.auction.Item" usage="read
        <class-cache class="org.hibernate.auction.Bid" usage="read-
        <collection-cache collection="org.hibernate.auction.Item.bi

    </session-factory>

</hibernate-configuration>
```

如你所见, 这个方法优势在于，在配置文件中指出了映射定义文件的名字. 一旦你需要调整Hibernate的缓存， `hibernate.cfg.xml` 也是更方便. 注意，使用 `hibernate.properties` 还是 `hibernate.cfg.xml` 完全是由你来决定, 除了上面提到的XML语法的优势之外, 两者是等价的.

使用XML配置，使得启动Hibernate变的异常简单, 如下所示，一行代码就可以搞定：

```
SessionFactory sf = new Configuration().configure().buildSessionFac
```

你可以使用如下代码来添加一个不同的XML配置文件

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

3.8. J2EE应用程序服务器的集成

针对J2EE体系,Hibernate有如下几个集成的方面:

- 容器管理的数据源*(Container-managed datasources)*: Hibernate能使用通过容器管理，并由JNDI提供的JDBC连接. 通常, 特别是当处理多个数据源的分布式事务的时候, 由一个JTA兼容的 `TransactionManager` 和一个 `ResourceManager` 来处理事务管理(CMT, 容器管理的事务). 当然你可以通过编程方式来划分事务边界(BMT, Bean管理的事务). 或者为了代码的可移植性，你也也许会想使用可选的 Hibernate `Transaction` API.
- 自动*JNDI*绑定: Hibernate可以在启动后将 `SessionFactory` 绑定到JNDI.
- *JTA Session*绑定*:* Hibernate `Session` 可以自动绑定到JTA事务作用的范围. 只需简单地从JNDI查找 `SessionFactory` 并获得当前的 `Session` . 当JTA事务完成时, 让Hibernate来处理 `Session` 的清洗(flush)与关闭. 事务的划分可以是声明式的(CMT),也可以是编程式的(BMT/UserTransaction).
- *JMX*部署*:* 如果你使用支持JMX应用程序服务器(如, JBoss AS), 那么你可以选择将Hibernate部署成托管MBean. 这将为你省去一行从 `Configuration` 构建 `SessionFactory` 的启动代码. 容器将启动你的 `HibernateService` , 并完美地处理好服务间的依赖关系 (在Hibernate启动前，数据源必须是可用的，等等).

如果应用程序服务器抛出"connection containment"异常, 根据你的环境，也许该将配置属性 `hibernate.connection.release_mode` 设为 `after_statement` .

3.8.1. 事务策略配置

在你的架构中，Hibernate的 `Session` API是独立于任何事务分界系统的. 如果你让Hibernate通过连接池直接使用JDBC, 你需要调用JDBC API来打开和关闭你的事务. 如果你运行在J2EE应用程序服务器中, 你也许想用Bean管理的事务并在需要的时候调用JTA API和 `UserTransaction` .

为了让你的代码在两种(或其他)环境中可以移植，我们建议使用可选的Hibernate `Transaction` API, 它包装并隐藏了底层系统. 你必须通过设置Hibernate配置属性 `hibernate.transaction.factory_class` 来指定 一个 `Transaction` 实例的工厂类.

有三个标准(内建)的选择:

`org.hibernate.transaction.JDBCTransactionFactory`

委托给数据库(JDBC)事务（默认）

`org.hibernate.transaction.JTATransactionFactory`

如果在上下文环境中存在运行着的事务(如, EJB会话Bean的方法), 则委托给容器管理的事务, 否则，将启动一个新的事务，并使用Bean管理的事务.

`org.hibernate.transaction.CMTTransactionFactory`

委托给容器管理的JTA事务

你也可以定义属于你自己的事务策略 (如, 针对CORBA的事务服务)

Hibernate的一些特性 (比如二级缓存, Contextual Sessions with JTA等等)需要访问在托管环境中的JTA `TransactionManager` . 由于J2EE没有标准化一个单一的机制,Hibernate在应用程序服务器中，你必须指定Hibernate如何获得 `TransactionManager` 的引用:

表 **3.10. JTA TransactionManagers**

Transaction工厂类
`org.hibernate.transaction.JBossTransactionManagerLookup`
`org.hibernate.transaction.WeblogicTransactionManagerLookup`
`org.hibernate.transaction.WebSphereTransactionManagerLookup`
`org.hibernate.transaction.WebSphereExtendedJTATransactionLookup`
`org.hibernate.transaction.OrionTransactionManagerLookup`
`org.hibernate.transaction.ResinTransactionManagerLookup`
`org.hibernate.transaction.JOTMTransactionManagerLookup`
`org.hibernate.transaction.JOnASTransactionManagerLookup`
`org.hibernate.transaction.JRun4TransactionManagerLookup`
`org.hibernate.transaction.BESTransactionManagerLookup`

3.8.2. JNDI绑定的 `SessionFactory`

与JNDI绑定的Hibernate的 `SessionFactory` 能简化工厂的查询，简化创建新的 `Session` . 需要注意的是这与JNDI绑定 `Datasource` 没有关系, 它们只是恰巧用了相同的注册表!

如果你希望将 `SessionFactory` 绑定到一个JNDI的名字空间, 用属性 `hibernate.session_factory_name` 指定一个名字(如, `java:hibernate/SessionFactory`). 如果不设置这个属性, `SessionFactory` 将不会被绑定到JNDI中. (在以只读JNDI为默认实现的环境中，这个设置尤其有用, 如Tomcat.)

在将 `SessionFactory` 绑定至JNDI时, Hibernate将使用 `hibernate.jndi.url` , 和 `hibernate.jndi.class` 的值来实例化初始环境(initial context). 如果它们没有被指定, 将使用默认的 `InitialContext` .

在你调用 `cfg.buildSessionFactory()` 后, Hibernate会自动将 `SessionFactory` 注册到JNDI. 这意味这你至少需要在你应用程序的启动代码(或工具类)中完成这个调用, 除非你使用 `HibernateService` 来做JMX部署 (见后面讨论).

假若你使用JNDI `SessionFactory` ,EJB或者任何其它类都可以从JNDI中找到此 `SessionFactory` 。

我们建议，在受管理的环境中，把 `SessionFactory` 绑定到JNDI，在其它情况下，使用一个 `static(静态的)` singleton。为了在你的应用程序代码中隐藏这些细节，我们还建议你用一个helper类把实际查找 `SessionFactory` 的代码隐藏起来，比如 `HibernateUtil.getSessionFactory()` 。注意，这个类也就可以方便地启动Hibernate，参见第一章。

3.8.3. 在JTA环境下使用Current Session context (当前session上下文)管理

在Hibernate中，管理 `Session` 和transaction最好的方法是自动的"当前" `Session` 管理。请参见第 2.5 节 “上下文相关的（Contextual）Session”一节的讨论。使用 `"jta"` session上下文，假若在当前JTA事务中还没有Hibernate `Session` 关联，第一次 `sessionFactory.getCurrentSession()` 调用会启动一个Session,并关联到当前的JTA事务。在 `"jta"` 上下文中调用 `getCurrentSession()` 获得的 `Session` ，会被设置为在transaction关闭的时候自动flush（清洗）、在transaction关闭之后自动关闭，每句语句之后主动释放JDBC连接。这就可以根据JTA事务的生命周期来管理与之关联的 `Session` ，用户代码中就可以不再考虑这些管理。你的代码也可以通过 `UserTransaction` 用编程方式使用JTA，或者(我们建议，为了便于移植代码）使用Hibernate的 `Transaction` API来设置transaction边界。如果你的代码运行在EJB容器中，建议对CMT使用声明式事务声明。

3.8.4. JMX部署

为了将 `SessionFactory` 注册到JNDI中， `cfg.buildSessionFactory()` 这行代码仍需在某处被执行. 你可在一个 `static` 初始化块(像 `HibernateUtil` 中的那样)中执行它或将Hibernate部署为一个托管的服务.

为了部署在一个支持JMX的应用程序服务器上，Hibernate和 `org.hibernate.jmx.HibernateService` 一同分发，如Jboss AS。 实际的部署和配置是由应用程序服务器提供者指定的. 这里是JBoss 4.0.x 的 `jboss-service.xml` 样例:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory'

    <!-- 必须的服务 -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- 将Hibernate服务绑定到JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attr

    <!-- 数据源设置 -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</at

    <!-- 事务集成 -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</at
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- 抓取选项 -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- 二级缓存 -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCach
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- 日志 -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- 映射定义文件 -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Cat

</mbean>

</server>
```

这个文件是部署在 `META-INF` 目录下的, 并会被打包到以 `.sar` (service archive)为扩展名的JAR文件中. 同时, 你需要将Hibernate、它所需要的第三方库、你编译好的持久化类以及你的映射定义文件打包进同一个文档. 你的企业Bean(一般为会话Bean)可能会被打包成它们自己的JAR文件, 但你也许会将EJB JAR文件一同包含进能独立(热)部署的主服务文档. 参考JBoss AS文档以了解更多的JMX服务与EJB部署的信息.

第 4 章 持久化类(Persistent Classes)

目录

在应用程序中，用来实现业务问题实体的（如，在电子商务应用程序中的Customer和Order） 类就是持久化类。不能认为所有的持久化类的实例都是持久的状态——一个实例的状态也可能 是瞬时的或脱管的。

如果这些持久化类遵循一些简单的规则，Hibernate能够工作得更好，这些规则也被称作 简单传统Java对象(POJO:Plain Old Java Object)编程模型。但是这些规则并不是必需的。 实际上，Hibernate3对于你的持久化类几乎不做任何设想。你可以用其他的方法来表达领域模型： 比如，使用 `Map` 实例的树型结构。

4.1. 一个简单的POJO例子

大多数Java程序需要用一个持久化类来表示猫科动物。

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
```

```
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

这里要遵循四条主要的规则：

4.1.1. 实现一个默认的（即无参数的）构造方法（constructor）

`Cat` 有一个无参数的构造方法。所有的持久化类都必须有一个 默认的构造方法（可以不是public的），这样的话Hibernate就可以使用 `Constructor.newInstance()` 来实例化它们。 我们强烈建议，在Hibernate中，为了运行期代理的生成，构造方法至少是 包*(package)*内可见的。

4.1.2. 提供一个标识属性（identifier property）（可选）

`Cat` 有一个属性叫做 `id` 。这个属性映射数据库表的主 键字段。这个属性可以叫任何名字，其类型可以是任何的原始类型、原始类型的包装类型、`java.lang.String` 或者是 `java.util.Date` 。（如果你的遗留数据库表有联合主键，你甚至可以用一个用户自定义的类，该类拥有这些类型 的属性。参见后面的关于联合标识符的章节。）

标识符属性是可选的。可以不用管它，让Hibernate内部来追踪对象的识别。 但是我们并不推荐这样做。

实际上，一些功能只对那些声明了标识符属性的类起作用：

- 托管对象的传播性再连接（级联更新或级联合并）——参阅 + 第 10.11 节 “传播性持久化(transitive persistence)”
- `Session.saveOrUpdate()`
- `Session.merge()`

我们建议你对持久化类声明命名一致的标识属性。我们还建议你使用一 个可以为空（也就是说，不是原始类型）的类型。

4.1.3. 使用非final的类 (可选)

代理（*proxies*）是Hibernate的一个重要的功能，它依赖的条件是，持久 化类或者是非final的，或者是实现了一个所有方法都声明为public的接口。

你可以用Hibernate持久化一个没有实现任何接口的 `final` 类，但是你 不能使用代理来延迟关联加载，这会限制你进行性能优化的选择。

你也应该避免在非final类中声明 `public final` 的方法。如果你想使用一 个有 `public final` 方法的类，你必须通过设置 `lazy="false"` 来明确地禁用代理。

4.1.4. 为持久化字段声明访问器(accessors)和是否可变的标志(mutators)(可选)

`Cat` 为它的所有持久化字段声明了访问方法。很多其他ORM工具直接对 实例变量进行持久化。我们相信，在关系数据库schema和类的内部数据结构之间引入间接层(原文为"非直接"，indirection)会好一些。默认情况下Hibernate持久化JavaBeans风格的属性，认可 `getFoo` ， `isFoo` 和 `setFoo` 这种形式的方法名。 如果需要，你可以对某些特定属性实行直接字段访问。

属性不需要要声明为public的。Hibernate可以持久化一个有 `default` 、 `protected` 或 `private` 的get/set方法对 的属性进行持久化。

TODO：property和proxy包里的用户扩展框架文档。

4.2. 实现继承（Inheritance）

子类也必须遵守第一条和第二条规则。它从超类 `Cat` 继承了标识属性。

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. 实现 `equals()` 和 `hashCode()`

如果你有如下需求，你必须重载 `equals()` 和 `hashCode()` 方法：

- 想把持久类的实例放入 `Set` 中（当表示多值关联时，推荐这么做）
- 想重用脱管实例

Hibernate保证，仅在特定会话范围内，持久化标识（数据库的行）和Java标识是等价的。因此，一旦 我们混合了从不同会话中获取的实例，如果希望 `Set` 有明确的语义，就必 须实现 `equals()` 和 `hashCode()` 。

实现 `equals()` / `hashCode()` 最显而易见的方法是比较两个对象 标识符的值。如果值相同，则两个对象对应于数据库的同一行，因此它们是相等的（如果都被添加到 `Set` ，则在 `Set` 中只有一个元素）。不幸的是，对生成的标识不能 使用这种方法。Hibernate仅对那些持久化对象赋标识值，一个新创建的实例将不会有任何标识值。此外， 如果一个实例没有被保存(unsaved)，并且它当前正在一个 `Set` 中，保存它将会给这个对象 赋一个标识值。如果 `equals()` 和 `hashCode()` 是基于标识值 实现的，则其哈希码将会改变，这违反了 `Set` 的契约。建议去Hibernate的站点阅读关于这个 问题的全部讨论。注意，这不是Hibernate的问题，而是一般的Java对象标识和Java对象等价的语义问题。

我们建议使用业务键值相等*(Business key equality)*来实现 `equals()` 和 `hashCode()` 。业务键值相等的意思是， `equals()` 方法 仅仅比较形成业务键的属性，它能在现实世界里标识我们的实例（是一个自然的候选码）。

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return fa
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

注意，业务键不必像数据库的主键那样固定不变（参见第 11.1.3 节 “关注对象标识 (Considering object identity)”）。 对业务键而言，不可变或唯一的属性是不错的选择。

4.4. 动态模型(Dynamic models)

注意，以下特性在当前处于试验阶段，将来可能会有变化。

运行期的持久化实体没有必要一定表示为像POJO类或JavaBean对象那样的形式。Hibernate也支持动态模型 （在运行期使用 `Map` 的 `Map` ）和象DOM4J的树模型那样的实体表示。使用这种方法，你不用写持久化类，只写映射文件就行了。

Hibernate默认工作在普通POJO模式。你可以使用配置选项 `default_entity_mode` ， 对特定的 `SessionFactory` ，设置一个默认的实体表示模式。 （参见表 3.3 “ Hibernate配置属性 ”。）

下面是用 `Map` 来表示的例子。首先，在映射文件中，要声明 `entity-name` 来代替一个类名（或作为一种附属）。

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

注意，虽然是用目标类名来声明关联的，但是关联的目标类型除了是POJO之外，也可以 是一个动态的实体。

在使用 `dynamic-map` 为 `SessionFactory` 设置了默认的实体模式之后，可以在运行期使用 `Map` 的 `Map` 。

```
Session s = openSession();
Transaction tx = s.beginTransaction();
Session s = openSession();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

动态映射的好处是，变化所需要的时间少了，因为原型不需要实现实体类。然而，你无法进行 编译期的类型检查，并可能由此会处理很多的运行期异常。幸亏有了Hibernate映射，它使得数 据库的schema能容易的规格化和合理化，并允许稍后在此之上添加合适的领域模型实现。

实体表示模式也能在每个 `Session` 的基础上设置：

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

请注意，用 `EntityMode` 调用 `getSession()` 是在 `Session` 的API中，而不是 `SessionFactory` 。这样，新的 `Session` 共享底层的JDBC连接，事务，和其他的上下文信 息。这意味着，你不需要在第二个 `Session` 中调用 `flush()` 和 `close()` ，同样的，把事务和连接的处理 交给原来的工作单元。

关于XML表示能力的更多信息可以在第 18 章 *XML*映射中找到。

4.5. 元组片断映射(Tuplizers)

`org.hibernate.tuple.Tuplizer`，以及其子接口，负责根据给定的`org.hibernate.EntityMode`，来复现片断数据。如果给定的片断数据被认为其是一种数据结构，"tuplizer"就是一个知道如何创建这样的数据结构，以及如何给这个数据结构赋值的东西。比如说，对于POJO这种Entity Mode，对应的tuplizer知道通过其构造方法来创建一个POJO，再通过其属性访问器来访问POJO属性。有两大类高层Tuplizer，分别是`org.hibernate.tuple.entity.EntityTuplizer`和`org.hibernate.tuple.entity.ComponentTuplizer`接口。`EntityTuplizer`负责管理上面提到的实体的契约，而`ComponentTuplizer`则是针对组件的。

用户也可以插入其自定义的tuplizer。或许您需要一种不同于dynamic-map entity-mode中使用的`java.util.HashMap`的`java.util.Map`实现；或许您需要与默认策略不同的代理生成策略(proxy generation strategy)。通过自定义tuplizer实现，这两个目标您都可以达到。Tuplizer定义被附加到它们期望管理的entity或者component映射中。回到我们的customer entity例子：

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>

public class CustomMapTuplizerImpl
        extends org.hibernate.tuple.entity.DynamicMapEntityTuplizer
    // override the buildInstantiator() method to plug in our custo
    protected final Instantiator buildInstantiator(
            org.hibernate.mapping.PersistentClass mappingInfo) {
        return new CustomMapInstantiator( mappingInfo );
    }

    private static final class CustomMapInstantiator
            extends org.hibernate.tuple.DynamicMapInstantitor {
        // override the generateMap() method to return our custom r
        protected final Map generateMap() {
            return new CustomMap();
        }
    }
}
```

第 5 章 对象/关系数据库映射基础(Basic O/R Mapping)

目录

5.1. 映射定义（Mapping declaration）

对象和关系数据库之间的映射通常是用一个XML文档(XML document)来定义的。这个映射文档被设计为易读的， 并且可以手工修改。映射语言是以Java为中心，这意味着映射文档是按照持久化类的定义来创建的， 而非表的定义。

请注意，虽然很多Hibernate用户选择手写XML映射文档，但也有一些工具可以用来生成映射文档， 包括XDoclet,Middlegen和AndroMDA。

让我们从一个映射的例子开始：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.d

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>
```

```
                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

我们现在开始讨论映射文档的内容。我们只描述Hibernate在运行时用到的文档元素和属性。 映射文档还包括一些额外的可选属性和元素，它们在使用schema导出工具的时候会影响导出的数据库schema结果。 （比如， `not-null` 属性。）

5.1.1. Doctype

所有的XML映射都需要定义如上所示的doctype。DTD可以从上述URL中获取， 也可以从 `hibernate-x.x.x/src/net/sf/hibernate` 目录中、或 `hibernate.jar` 文件中找到。Hibernate总是会首先在它的classptah中搜索DTD文件。 如果你发现它是通过连接Internet查找DTD文件，就对照你的classpath目录检查XML文件里的DTD声明。

5.1.1.1. EntityResolver

As mentioned previously, Hibernate will first attempt to resolve DTDs in its classpath. The manner in which it does this is by registering a custom `org.xml.sax.EntityResolver` implementation with the SAXReader it uses to read in the xml files. This custom `EntityResolver` recognizes two different systemId namespaces. 如前所述,Hibernate首先在其classpath中查找DTD。其行为是依靠在系统中注册的 `org.xml.sax.EntityResolver` 的一个具体实现，SAXReader依靠它来读取xml文件。这一 `EntityResolver` 实现能辨认两种不同的 systenId命名空间。

- 若resolver遇到了一个以 `http://hibernate.sourceforge.net/` 为开头的systemId，它会辨认出是 `hibernate namespace` ，resolver就试图通过加载Hibernate类的classloader来查找这些实体。
- 若resolver遇到了一个使用 `classpath://` URL协议的systemId，它会辨认出这是 `user namespace` ,resolver试图通过(1)当前线程上下文的classloader和(2)加载Hibernate class的classloader来查找这些实体。

使用user namespace(用户命名空间)的例子：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dt
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

`types.xml` 是 `your.domain` 包中的一个资源，它包含了一个定制的第 5.2.3 节“自定义值类型”。

5.1.2. hibernate-mapping

这个元素包括一些可选的属性。 `schema` 和 `catalog` 属性， 指明了这个映射所连接（refer）的表所在的schema和/或catalog名称。 假若指定了这个属性，表名会加上所指定的schema和catalog的名字扩展为全限定名。假若没有指定，表名就不会使用全限定名。 `default-cascade` 指定了未明确注明 `cascade` 属性的Java属性和 集合类Hibernate会采取什么样的默认级联风格。 `auto-import` 属性默认让我们在查询语言中可以使用 非全限定名的类名。

```
<hibernate-mapping
         schema="schemaName"
         catalog="catalogName"
         default-cascade="cascade_style"
         default-access="field|property|ClassName"
         default-lazy="true|false"
         auto-import="true|false"
         package="package.name"
 />
```

❶	**`schema` (可选): 数据库schema的名称。**
❷	`catalog` (可选): 数据库catalog的名称。
❸	`default-cascade` (可选 - 默认为 `none`): 默认的级联风格。
❹	`default-access` (可选 - 默认为 `property`): Hibernate用来访问所有属性的策略。可以通过实现 `PropertyAccessor` 接口 自定义。
❺	`default-lazy` (可选 - 默认为 `true`): 指定了未明确注明 `lazy` 属性的Java属性和集合类， Hibernate会采取什么样的默认加载风格。
❻	`auto-import` (可选 - 默认为 `true`): 指定我们是否可以在查询语言中使用非全限定的类名（仅限于本映射文件中的类）。
❼	`package` (可选): 指定一个包前缀，如果在映射文档中没有指定全限定的类名， 就使用这个作为包名。

假若你有两个持久化类，它们的非全限定名是一样的（就是两个类的名字一样，所在的包不一样--译者注）， 你应该设置 `auto-import="false"` 。如果你把一个“import过”的名字同时对应两个类， Hibernate会抛出一个异常。

注意 `hibernate-mapping` 元素允许你嵌套多个如上所示的 `<class>` 映射。但是最好的做法（也许一些工具需要的）是一个 持久化类（或一个类的继承层次）对应一个映射文件，并以持久化的超类名称命名，例如： `Cat.hbm.xml` ， `Dog.hbm.xml` ，或者如果使用继承， `Animal.hbm.xml` 。

5.1.3. class

你可以使用 `class` 元素来定义一个持久化类：

```
<class
        name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        mutable="true|false"
        schema="owner"
        catalog="catalog"
        proxy="ProxyInterface"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        select-before-update="true|false"
        polymorphism="implicit|explicit"
        where="arbitrary sql where condition"
        persister="PersisterClass"
        batch-size="N"
        optimistic-lock="none|version|dirty|all"
        lazy="true|false"
        entity-name="EntityName"
        check="arbitrary sql check condition"
        rowid="rowid"
        subselect="SQL expression"
        abstract="true|false"
        node="element-name"
/>
```

❶	**name (可选): 持久化类（或者接口）的Java全限定名。如果这个属性不存在，Hibernate将假定这是一个非POJO的实体映射。**
❷	`table` (可选 - 默认是类的非全限定名): 对应的数据库表名。
❸	`discriminator-value` (可选 - 默认和类名一样): 一个用于区分不同的子类的值，在多态行为时使用。它可以接受的值包括 `null` 和 `not null` 。
❹	`mutable` (可选，默认值为 `true`): 表明该类的实例是可变的或者不可变的。
❺	`schema` (可选): 覆盖在根 `<hibernate-mapping>` 元素中指定的schema名字。
❻	`catalog` (可选): 覆盖在根 `<hibernate-mapping>` 元素中指定的catalog名字。
❼	`proxy` (可选): 指定一个接口，在延迟装载时作为代理使用。 你可以在

	这里使用该类自己的名字。
❽	`dynamic-update` (可选, 默认为 `false`): 指定用于 `UPDATE` 的SQL 将会在运行时动态生成，并且只更新那些改变过的字段。
❾	`dynamic-insert` (可选, 默认为 `false`): 指定用于 `INSERT` 的 SQL 将会在运行时动态生成，并且只包含那些非空值字段。
❿	`select-before-update` (可选, 默认为 `false`): 指定Hibernate除非确定对象真正被修改了（如果该值为true－译注），否则不会执行SQL `UPDATE` 操作。在特定场合（实际上，它只在一个瞬时对象（transient object）关联到一个 新的session中时执行的update()中生效），这说明Hibernate会在 `UPDATE` 之前执行一次额外的SQL `SELECT` 操作，来决定是否应该执行 `UPDATE` 。
⓫	`polymorphism（多态）` (可选, 默认值为 `implicit (隐式)`): 界定是隐式还是显式的使用多态查询（这只在Hibernate的具体表继承策略中用到－译注）。
⓬	`where` (可选) 指定一个附加的SQL `WHERE` 条件， 在抓取这个类的对象时会一直增加这个条件。
⓭	`persister` (可选): 指定一个定制的 `ClassPersister` 。
⓮	`batch-size` (可选,默认是 `1`) 指定一个用于 根据标识符（identifier）抓取实例时使用的"batch size"（批次抓取数量）。
⓯	`optimistic-lock（乐观锁定）` (可选，默认是 `version`): 决定乐观锁定的策略。
(16)	`lazy` (可选): 通过设置 `lazy="false"` ， 所有的延迟加载（Lazy fetching）功能将被全部禁用（disabled）。
(17)	`entity-name` (可选，默认为类名): Hibernate3允许一个类进行多次映射（ 前提是映射到不同的表），并且允许使用Maps或XML代替Java层次的实体映射 （也就是实现动态领域模型，不用写持久化类－译注）。更多信息请看第 4.4 节 “动态模型(Dynamic models)” and 第 18 章 *XML映射*。
(18)	`check` (可选): 这是一个SQL表达式， 用于为自动生成的schema添加多行（multi-row）约束检查。
(19)	`rowid` (可选): Hibernate可以使用数据库支持的所谓的ROWIDs，例如： Oracle数据库，如果你设置这个可选的 `rowid` ， Hibernate可以使用额外的字段 `rowid` 实现快速更新。ROWID是这个功能实现的重点， 它代表了一个存储元组（tuple）的物理位置。
(20)	`subselect` (可选): 它将一个不可变（immutable）并且只读的实体映射到一个数据库的 子查询中。当你想用视图代替一张基本表的时候，这是有用的，但最好不要这样做。更多的介绍请看下面内容。
	`abstract` (可选): 用于在 `<union-subclass>` 的继承结构

(21)	（hierarchies）中标识抽象超类。

若指明的持久化类实际上是一个接口，这也是完全可以接受的。 之后你可以用元素 `<subclass>` 来指定该接口的实际实现类。 你可以持久化任何*static*（静态的）内部类。 你应该使用标准的类名格式来指定类名， `比如：Foo$Bar` 。

不可变类， `mutable="false"` 不可以被应用程序更新或者删除。 这可以让Hibernate做一些小小的性能优化。

可选的 `proxy` 属性允许延迟加载类的持久化实例。 Hibernate开始会返回实现了这个命名接口的CGLIB代理。当代理的某个方法被实际调用的时候， 真实的持久化对象才会被装载。参见下面的“用于延迟装载的代理”。

Implicit (隐式)的多态是指，如果查询时给出的是任何超类、该类实现的接口或者该类的 名字，都会返回这个类的实例；如果查询中给出的是子类的名字，则会返回子类的实例。 *Explicit* （显式）的多态是指，只有在查询时给出明确的该类名字时才会返回这个类的实例； 同时只有在这个 `<class>` 的定义中作为 `<subclass>` 或者 `<joined-subclass>` 出现的子类，才会可能返回。 在大多数情况下，默认的 `polymorphism="implicit"` 都是合适的。 显式的多态在有两个不同的类映射到同一个表的时候很有用。（允许一个“轻型”的类，只包含部分表字段）。

`persister` 属性可以让你定制这个类使用的持久化策略。 你可以指定你自己实现 `org.hibernate.persister.EntityPersister` 的子类，你甚至可以完全从头开始编写一个 `org.hibernate.persister.ClassPersister` 接口的实现， 比如是用储存过程调用、序列化到文件或者LDAP数据库来实现。 参阅 `org.hibernate.test.CustomPersister` ，这是一个简单的例子 （“持久化”到一个 `Hashtable` ）。

请注意 `dynamic-update` 和 `dynamic-insert` 的设置并不会继承到子类， 所以在 `<subclass>` 或者 `<joined-subclass>` 元素中可能 需要再次设置。这些设置是否能够提高效率要视情形而定。请用你的智慧决定是否使用。

使用 `select-before-update` 通常会降低性能。如果你重新连接一个脱管（detache）对象实例 到一个 `Session` 中时，它可以防止数据库不必要的触发update。 这就很有用了。

如果你打开了 `dynamic-update` ，你可以选择几种乐观锁定的策略：

- `version（版本检查）` 检查version/timestamp字段
- `all（全部）` 检查全部字段
- `dirty（脏检查）` 只检察修改过的字段
- `none（不检查）` 不使用乐观锁定

我们非常强烈建议你在Hibernate中使用version/timestamp字段来进行乐观锁定。 对性能来说，这是最好的选择，并且这也是唯一能够处理在session外进行操作的策略（例如：在使用 `Session.merge()` 的时候）。

对Hibernate映射来说视图和表是没有区别的，这是因为它们在数据层都是透明的（注意：一些数据库不支持视图属性，特别是更新的时候）。有时你想使用视图，但却不能在数据库 中创建它（例如：在遗留的schema中）。这样的话，你可以映射一个不可变的（immutable）并且是 只读的实体到一个给定的SQL子查询表达式：

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

定义这个实体用到的表为同步（synchronize），确保自动刷新（auto-flush）正确执行， 并且依赖原实体的查询不会返回过期数据。 <subselect> 在属性元素 和一个嵌套映射元素中都可见。

5.1.4. id

被映射的类必须定义对应数据库表主键字段。大多数类有一个JavaBeans风格的属性， 为每一个实例包含唯一的标识。 `<id>` 元素定义了该属性到数据库表主键字段的映射。

```
<id
        name="propertyName"
        type="typename"
        column="column_name"
        unsaved-value="null|any|none|undefined|id_value"
        access="field|property|ClassName"
        node="element-name|@attribute-name|element/@attribute|.">

        <generator class="generatorClass"/>
</id>
```

❶	**`name` (可选):** 标识属性的名字。
❷	`type` (可选): 标识Hibernate类型的名字。
❸	`column` (可选 - 默认为属性名): 主键字段的名字。
❹	`unsaved-value` (可选 - 默认为一个切合实际（sensible）的值): 一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。 这可以把这种实例和从以前的session中装载过（可能又做过修改--译者注） 但未再次持久化的实例区分开来。
❺	`access` (可选 - 默认为 `property`): Hibernate用来访问属性值的策略。

如果 `name` 属性不存在，会认为这个类没有标识属性。

`unsaved-value` 属性在Hibernate3中几乎不再需要。

还有一个另外的 `<composite-id>` 定义可以访问旧式的多主键数据。 我们强烈不建议使用这种方式。

5.1.4.1. Generator

可选的 `<generator>` 子元素是一个Java类的名字， 用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数， 用 `<param>` 元素来传递。

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

所有的生成器都实现 `org.hibernate.id.IdentifierGenerator` 接口。 这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然， Hibernate提供了很多内置的实现。下面是一些内置生成器的快捷名字：

`increment`

用于为 `long` , `short` 或者 `int` 类型生成 唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。 在集群下不要使用。

`identity`

对DB2,MySQL, MS SQL Server, Sybase和HypersonicSQL的内置标识字段提供支持。 返回的标识符是 `long` , `short` 或者 `int` 类型的。

`sequence`

在DB2,PostgreSQL, Oracle, SAP DB, McKoi中使用序列（sequence)， 而在Interbase中使用生成器(generator)。返回的标识符是 `long` , `short` 或者 `int` 类型的。

`hilo`

使用一个高/低位算法高效的生成 `long` , `short` 或者 `int` 类型的标识符。给定一个表和字段（默认分别是 `hibernate_unique_key` 和 `next_hi` ）作为高位值的来源。 高/低位算法生成的标识符只在一个特定的数据库中是唯一的。

`seqhilo`

使用一个高/低位算法来高效的生成 `long` , `short` 或者 `int` 类型的标识符，给定一个数据库序列（sequence)的名字。

`uuid`

用一个128-bit的UUID算法生成字符串类型的标识符， 这在一个网络中是唯一的（使用了IP地址）。UUID被编码为一个32位16进制数字的字符串。

`guid`

在MS SQL Server 和 MySQL 中使用数据库生成的GUID字符串。

`native`

根据底层数据库的能力选择 `identity`, `sequence` 或者 `hilo` 中的一个。

`assigned`

让应用程序在 `save()` 之前为对象分配一个标示符。这是 `<generator>` 元素没有指定时的默认生成策略。

`select`

通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。

`foreign`

使用另外一个相关联的对象的标识符。通常和 `<one-to-one>` 联合起来使用。

`sequence-identity`

一种特别的序列生成策略,使用数据库序列来生成实际值,但将它和JDBC3的getGeneratedKeys结合在一起,使得在插入语句执行的时候就返回生成的值。目前为止只有面向JDK 1.4的Oracle 10g驱动支持这一策略。注意，因为Oracle驱动程序的一个bug，这些插入语句的注释被关闭了。（原文：Note comments on these insert statements are disabled due to a bug in the Oracle drivers.）

5.1.4.2. 高/低位算法（Hi/Lo Algorithm）

`hilo` 和 `seqhilo` 生成器给出了两种hi/lo算法的实现， 这是一种很令人满意的标识符生成算法。第一种实现需要一个“特殊”的数据库表来保存下一个可用的“hi”值。 第二种实现使用一个Oracle风格的序列（在被支持的情况下）。

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

很不幸，你在为Hibernate自行提供 `Connection` 时无法使用 `hilo` 。 当Hibernate使用JTA获取应用服务器的数据源连接时,你必须正确地配置 `hibernate.transaction.manager_lookup_class` 。

5.1.4.3. UUID算法（UUID Algorithm ）

UUID包含：IP地址，JVM的启动时间（精确到1/4秒），系统时间和一个计数器值（在JVM中唯一）。 在Java代码中不可能获得MAC地址或者内存地址，所以这已经是我们在不使用JNI的前提下的能做的最好实现了。

5.1.4.4. 标识字段和序列（Identity columns and Sequences）

对于内部支持标识字段的数据库(DB2,MySQL,Sybase,MS SQL)，你可以使用 `identity` 关键字生成。 对于内部支持序列的数据库（DB2,Oracle, PostgreSQL, Interbase, McKoi,SAP DB), 你可以使用 `sequence` 风格的关键字生成。 这两种方式对于插入一个新的对象都需要两次SQL查询。

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

对于跨平台开发， `native` 策略会从 `identity` ， `sequence` 和 `hilo` 中进行选择，选择哪一个，这取决于底层数据库的支持能力。

5.1.4.5. 程序分配的标识符（Assigned Identifiers）

如果你需要应用程序分配一个标示符（而非Hibernate来生成），你可以使用 `assigned` 生成器。这种特殊的生成器会使用已经分配给对象的标识符属性的标识符值。 这个生成器使用一个自然键（natural key，有商业意义的列－译注）作为主键，而不是使用一个代理键（ surrogate key，没有商业意义的列－译注）。这是没有指定 `<generator>` 元素时的默认行为

当选择 `assigned` 生成器时，除非有一个version或timestamp属性，或者你定义了 `Interceptor.isUnsaved()` ，否则需要让Hiberante使用 `unsaved-value="undefined"` ，强制Hibernatet查询数据库来确定一个实例是瞬时的（transient） 还是脱管的（detached）。

5.1.4.6. 触发器实现的主键生成器（Primary keys assigned by triggers）

仅仅用于遗留的schema中 (Hibernate不能使用触发器生成DDL)。

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

在上面的例子中，类定义了一个命名为 `socialSecurityNumber` 的唯一值属性，它是一个自然键（natural key），命名为 `person_id` 的代理键（surrogate key）的值由触发器生成。

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName"
        node="element-name|."
        >

        <key-property name="propertyName" type="typename" column="c
        <key-many-to-one name="propertyName class="ClassName" colum
        ......
</composite-id>
```

如果表使用联合主键，你可以映射类的多个属性为标识符属性。 `<composite-id>` 元素接受 `<key-property>` 属性映射和 `<key-many-to-one>` 属性映射作为子元素。

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

你的持久化类必须重载 `equals()` 和 `hashCode()` 方法，来实现组合的标识符的相等判断。 实现 `Serializable` 接口也是必须的。

不幸的是，这种组合关键字的方法意味着一个持久化类是它自己的标识。除了对象自己之外， 没有什么方便的“把手”可用。你必须初始化持久化类的实例，填充它的标识符属性，再 `load()` 组合关键字关联的持久状态。我们把这种方法称为 *embedded*（嵌入式）的组合标识符，在重要的应用中不鼓励使用这种用法。

第二种方法我们称为*mapped(*映射式*)*组合标识符 (mapped composite identifier), `<composite-id>` 元素中列出的标识属性不但在持久化类出现，还形成一个独立的标识符类。

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

在这个例子中，组合标识符类 `MedicareId` 和实体类都含有 `medicareNumber` 和 `dependent` 属性。标识符类必须重载 `equals()` 和 `hashCode()` 并且实现 `Serializable` 接口。这种方法的缺点是出现了明显的代码重复。

下面列出的属性是用来指定一个映射式组合标识符的：

- `mapped` (可选，默认为 `false`): 指明使用一个映射式组合标识符，其包含的属性映射同时在实体类和组合标识符类中出现。
- `class` (可选,但对映射式组合标识符必须指定): 作为组合标识符类使用的类名.

在第 8.4 节 “组件作为联合标识符(Components as composite identifiers)”一节中,我们会描述第三种方式,那就是把组合标识符实现为一个组件(component)类,这是更方便的方法。下面的属性仅对第三种方法有效：

- `name` (可选,但对这种方法而言必须): 包含此组件标识符的组件类型的名字(参阅第9章).
- `access` (可选 - 默认为 `property`): Hibernate应该使用的访问此属性值的策略
- `class` (可选 - 默认会用反射来自动判定属性类型): 用来作为组合标识符的组件类的类名（参阅下一节）

第三种方式，被称为*identifier component(标识符组件)*是我们对几乎所有应用都推荐使用的方式。

5.1.6. 鉴别器（discriminator）

在"一棵对象继承树对应一个表"的策略中，`<discriminator>` 元素是必需的，它定义了表的鉴别器字段。鉴别器字段包含标志值，用于告知持久化层应该为某个特定的行创建哪一个子类的实例。 如下这些受到限制的类型可以使用： `string` , `character` , `integer` , `byte` , `short` , `boolean` , `yes_no` , `true_false` .

```
<discriminator
        column="discriminator_column"
        type="discriminator_type"
        force="true|false"
        insert="true|false"
        formula="arbitrary sql expression"
/>
```

❶	**`column`** (可选 - 默认为 **`class`**) 鉴别器字段的名字
❷	`type` (可选 - 默认为 `string`) 一个Hibernate字段类型的名字
❸	`force(强制)` (可选 - 默认为 `false`) "强制"Hibernate指定允许的鉴别器值,即使当取得的所有实例都是根类的。
❹	`insert` (可选 - 默认为 `true`) 如果你的鉴别器字段也是映射为复合标识（composite identifier）的一部分，则需将 这个值设为 `false` 。（告诉Hibernate在做SQL `INSERT` 时不包含该列）
❺	`formula` (可选) 一个SQL表达式，在类型判断（判断是父类还是具体子类一译注）时执行。可用于基于内容的鉴别器。

鉴别器字段的实际值是根据 `<class>` 和 `<subclass>` 元素中的 `discriminator-value` 属性得来的。

`force` 属性仅仅在这种情况下有用的：表中包含没有被映射到持久化类的附加辨别器值。 这种情况不会经常遇到。

使用 `formula` 属性你可以定义一个SQL表达式，用来判断一个行数据的类型。

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1
    type="integer"/>
```

5.1.7. 版本（version）(可选)

`<version>` 元素是可选的，表明表中包含附带版本信息的数据。 这在你准备使用 长事务（*long transactions*）的时候特别有用。（见后）

```
<version
        column="version_column"
        name="propertyName"
        type="typename"
        access="field|property|ClassName"
        unsaved-value="null|negative|undefined"
        generated="never|always"
        insert="true|false"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

❶	**`column` (可选 - 默认为属性名): 指定持有版本号的字段名。**
❷	`name` : 持久化类的属性名。
❸	`type` (可选 - 默认是 `integer`): 版本号的类型。
❹	`access` (可选 - 默认是 `property`): Hibernate用于访问属性值的策略。
❺	`unsaved-value` (可选 - 默认是 `undefined`): 用于标明某个实例时刚刚被实例化的（尚未保存）版本属性值，依靠这个值就可以把这种情况 和已经在先前的session中保存或装载的脱管（detached）实例区分开来。（ `undefined` 指明应被使用的标识属性值。）
❻	`generated` (可选 - 默认是 `never`): 表明此版本属性值是否实际上是由数据库生成的。请参阅第 5.6 节 “数据库生成属性（Generated Properties）”部分的讨论。
❼	`insert` (可选 - 默认是 `true`): 表明此版本列应该包含在SQL插入语句中。只有当数据库字段有默认值 `0` 的时候，才可以设置为 `false` 。

版本号必须是以下类型： `long` , `integer` , `short` , `timestamp` 或者 `calendar` 。

一个脱管（detached）实例的version或timestamp属性不能为空（null），因为Hibernate不管 `unsaved-value` 被指定为何种策略，它将任何属性为空的version或timestamp 实例看作为瞬时（transient）实例。 避免*Hibernate*中的传递重附（*transitive reattachment*）问题的一个简单方法是 定义一个不能为空的*version*或*timestamp*属性，特别是在人们使用程序分配的标识符（*assigned identifiers*） 或复合主键时非常有用！

5.1.8. timestamp (可选)

可选的 `<timestamp>` 元素指明了表中包含时间戳数据。 这用来作为版本的替代。时间戳本质上是一种对乐观锁定的一种不是特别安全的实现。当然， 有时候应用程序可能在其他方面使用时间戳。

```
<timestamp
        column="timestamp_column"
        name="propertyName"
        access="field|property|ClassName"
        unsaved-value="null|undefined"
        source="vm|db"
        generated="never|always"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

❶	**`column`** (可选 - 默认为属性名): 持有时间戳的字段名。
❷	`name` : 在持久化类中的JavaBeans风格的属性名， 其Java类型是 `Date` 或者 `Timestamp` 的。
❸	`access` (可选 - 默认是 `property`): Hibernate用于访问属性值的策略。
❹	`unsaved-value` (可选 - 默认是 `null`): 用于标明某个实例时刚刚被实例化的（尚未保存）版本属性值，依靠这个值就可以把这种情况和 已经在先前的session中保存或装载的脱管（detached）实例区分开来。 （ `undefined` 指明使用标识属性值进行这种判断。）
❺	`source` (可选 - 默认是 `vm`): Hibernate如何才能获取到时间戳的值呢？从数据库，还是当前JVM？从数据库获取会带来一些负担，因为Hibernate必须访问数据库来获得“下一个值”，但是在集群环境中会更安全些。还要注意，并不是所有的 `Dialect`（方言） 都支持获得数据库的当前时间戳的，而支持的数据库中又有一部分因为精度不足,用于锁定是不安全的（例如Oracle 8）。
❻	`generated` (可选 - 默认是 `never`): 指出时间戳值是否实际上是由数据库生成的.请参阅第 5.6 节 “数据库生成属性（Generated Properties）”的讨论。

注意， `<timestamp>` 和 `<version type="timestamp">` 是等价的。并且 `<timestamp source="db">` 和 `<version type="dbtimestamp"&gt` 是等价的。

5.1.9. property

<property> 元素为类定义了一个持久化的,JavaBean风格的属性。

```
<property
        name="propertyName"
        column="column_name"
        type="typename"
        update="true|false"
        insert="true|false"
        formula="arbitrary SQL expression"
        access="field|property|ClassName"
        lazy="true|false"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        generated="never|insert|always"
        node="element-name|@attribute-name|element/@attribute|."

        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

❶	`name`：属性的名字,以小写字母开头。
❷	`column` (可选 - 默认为属性名字): 对应的数据库字段名。 也可以通过嵌套的 `<column>` 元素指定。
❸	`type` (可选): 一个Hibernate类型的名字。
❹	`update, insert` (可选 - 默认为 `true`) : 表明用于 `UPDATE` 和/或 `INSERT` 的SQL语句中是否包含这个被映射了的字段。这二者如果都设置为 `false` 则表明这是一个"外源性（derived）"的属性，它的值来源于映射到同一个（或多个） 字段的某些其他属性，或者通过一个trigger(触发器）或其他程序生成。
❺	`formula` (可选): 一个SQL表达式，定义了这个计算 （*computed*） 属性的值。计算属性没有和它对应的数据库字段。
❻	`access` (可选 - 默认值为 `property`): Hibernate用来访问属性值的策略。
❼	`lazy` (可选 - 默认为 `false`): 指定 指定实例变量第一次被访问时，这个属性是否延迟抓取（fetched lazily）（ 需要运行时字节码增强）。
❽	`unique` (可选): 使用DDL为该字段添加唯一的约束。 同样，允许它作为 `property-ref` 引用的目标。
❾	`not-null` (可选): 使用DDL为该字段添加可否为空（nullability）的约束。
❿	`optimistic-lock` (可选 - 默认为 `true`): 指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。 换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。
⓫	`generated` (可选 - 默认为 `never`): 表明此属性值是否实际上是由数据库生成的。请参阅第 5.6 节 "数据库生成属性（Generated Properties）"的讨论。

*typename*可以是如下几种：

1. Hibernate基本类型名（比如： `integer, string, character,date, timestamp, float, binary, s` ）。
2. 一个Java类的名字，这个类属于一种默认基础类型 (比如： `int, float,char, java.lang.String, java.util.Date, java.lang.Int`)。
3. 一个可以序列化的Java类的名字。
4. 一个自定义类型的类的名字。（比如： `com.illflow.type.MyCustomType`)。

如果你没有指定类型，Hibernarte会使用反射来得到这个名字的属性，以此来猜测正确的Hibernate类型。 Hibernate会按照规则2,3,4的顺序对属性读取器(getter方法）的返回类进行解释。然而，这还不够。 在某些情况下你仍然需要 `type` 属性。（比如，为了区别 `Hibernate.DATE` 和 `Hibernate.TIMESTAMP` ,或者为了指定一个自定义类型。）

`access` 属性用来让你控制Hibernate如何在运行时访问属性。在默认情况下，Hibernate会使用属性的get/set方法对（pair）。如果你指明 `access="field"` ，Hibernate会忽略get/set方法对，直接使用反射来访问成员变量。你也可以指定你自己的策略， 这就需要你自己实现 `org.hibernate.property.PropertyAccessor` 接口， 再在access中设置你自定义策略类的名字。

衍生属性（derive propertie）是一个特别强大的特征。这些属性应该定义为只读，属性值在装载时计算生成。 你用一个SQL表达式生成计算的结果，它会在这个实例转载时翻译成一个SQL查询的 `SELECT` 子查询语句。

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, 
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

注意，你可以使用实体自己的表，而不用为这个特别的列定义别名（ 上面例子中的 `customerId` ）。同时注意，如果你不喜欢使用属性， 你可以使用嵌套的 `<formula>` 映射元素。

5.1.10. 多对一（many-to-one）

通过 `many-to-one` 元素,可以定义一种常见的与另一个持久化类的关联。 这种关系模型是多对一关联（实际上是一个对象引用－译注）：这个表的一个外键引用目标表的 主键字段。

```
<many-to-one
        name="propertyName"
        column="column_name"
        class="ClassName"
        cascade="cascade_style"
        fetch="join|select"
        update="true|false"
        insert="true|false"
        property-ref="propertyNameFromAssociatedClass"
        access="field|property|ClassName"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        lazy="proxy|no-proxy|false"
        not-found="ignore|exception"
        entity-name="EntityName"
        formula="arbitrary SQL expression"
        node="element-name|@attribute-name|element/@attribute|."

        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"

/>
```

❶	`name` : 属性名。
❷	`column` (可选): 外间字段名。它也可以通过嵌套的 `<column>` 元素指定。
❸	`class` (可选 - 默认是通过反射得到属性类型): 关联的类的名字。
❹	`cascade（级联）` (可选): 指明哪些操作会从父对象级联到关联的对象。
❺	`fetch` (可选 - 默认为 `select`): 在外连接抓取（outer-join fetching）和序列选择抓取（sequential select fetching）两者中选择其一。
❻	`update, insert` (可选 - 默认为 `true`) 指定对应的字段是否包含在用于 `UPDATE` 和/或 `INSERT` 的SQL语句中。如果二者都是 `false` ,则这是一个纯粹的 “外源性（derived）”关联，它的值是通过映射到同一个（或多个）字段的某些其他属性得到 或者通过trigger(触发器）、或其他程序生成。
❻	`property-ref` : (可选) 指定关联类的一个属性，这个属性将会和本外键相对应。 如果没有指定，会使用对方关联类的主键。
❼	`access` (可选 - 默认是 `property`): Hibernate用来访问属性的策略。
❽	`unique` (可选): 使用DDL为外键字段生成一个唯一约束。此外， 这也可以用作 `property-ref` 的目标属性。这使关联同时具有 一对一的效果。
❾	`not-null` (可选): 使用DDL为外键字段生成一个非空约束。
❿	`optimistic-lock` (可选 - 默认为 `true`): 指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。 换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。
⓫	`lazy` (可选 - 默认为 `proxy`): 默认情况下，单点关联是经过代理的。 `lazy="no-proxy"` 指定此属性应该在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节码的增强）。 `lazy="false"` 指定此关联总是被预先抓取。
⓬	`not-found` (可选 - 默认为 `exception`): 指定外键引用的数据不存在时如何处理： `ignore` 会将行数据不存在视为一个空（null）关联。
⓭	`entity-name` (可选): 被关联的类的实体名。
⓮	`formula` (可选): SQL表达式，用于定义*computed*（计算出的）外键值。

`cascade` 属性设置为除了 `none` 以外任何有意义的值， 它将把特定的操作传递到关联对象中。这个值就代表着Hibernate基本操作的名称， `persist, merge, delete, save-update, evict, replicate, lock, refresh` ， 以及特别的值 `delete-orphan` 和 `all` ，并且可以用逗号分隔符 来组合这些操作，例如， `cascade="persist,merge,evict"` 或 `cascade="all,delete-orphan"` 。更全面的解释请参考第 10.11 节 “传播性持久化(transitive persistence)”. 注意， 单值关联 (many-to-one 和 one-to-one 关联) 不支持删除孤儿（orphan delete，删除不再被引用的值）.

一个典型的简单 `many-to-one` 定义例子：

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

`property-ref` 属性只应该用来对付遗留下来的数据库系统， 可能有外键指向对方关联表的是个非主键字段（但是应该是一个惟一关键字）的情况下。 这是一种十分丑陋的关系模型。比如说，假设 `Product` 类有一个惟一的序列号， 它并不是主键。（ `unique` 属性控制Hibernate通过SchemaExport工具进行的DDL生成。）

```
<property name="serialNumber" unique="true" type="string" column="S
```

那么关于 `OrderItem` 的映射可能是：

```
<many-to-one name="product" property-ref="serialNumber" column="PRO
```

当然，我们决不鼓励这种用法。

如果被引用的唯一主键由关联实体的多个属性组成，你应该在名称为 `<properties>` 的元素 里面映射所有关联的属性。

假若被引用的唯一主键是组件的属性，你可以指定属性路径：

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER
```

5.1.11. 一对一

持久化对象之间一对一的关联关系是通过 `one-to-one` 元素定义的。

```
<one-to-one
        name="propertyName"
        class="ClassName"
        cascade="cascade_style"
        constrained="true|false"
        fetch="join|select"
        property-ref="propertyNameFromAssociatedClass"
        access="field|property|ClassName"
        formula="any SQL expression"
        lazy="proxy|no-proxy|false"
        entity-name="EntityName"
        node="element-name|@attribute-name|element/@attribute|."

        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

❶	**name** : 属性的名字。
❷	class (可选 - 默认是通过反射得到的属性类型) : 被关联的类的名字。
❸	cascade(级联) (可选) 表明操作是否从父对象级联到被关联的对象。
❹	constrained(约束) (可选) 表明该类对应的表对应的数据库表，和被关联的对象所对应的数据库表之间，通过一个外键引用对主键进行约束。 这个选项影响 save() 和 delete() 在级联执行时的先后顺序以及 决定该关联能否被委托(也在schema export tool中被使用).
❺	fetch (可选 - 默认设置为 选择): 在外连接抓取或者序列选择抓取选择其一.
❻	property-ref : (可选) 指定关联类的属性名，这个属性将会和本类的主键相对应。如果没有指定，会使用对方关联类的主键。
❼	access (可选 - 默认是 property): Hibernate用来访问属性的策略。
❽	formula (可选):绝大多数一对一的关联都指向其实体的主键。在一些少见的情况中， 你可能会指向其他的一个或多个字段，或者是一个表达式，这些情况下，你可以用一个SQL公式来表示。 （可以在org.hibernate.test.onetooneformula找到例子）
❾	lazy (可选 - 默认为 proxy): 默认情况下，单点关联是经过代理的。 lazy="no-proxy" 指定此属性应该在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节码的增强）。 lazy="false" 指定此关联总是被预先抓取。注意，如果 *constrained="false"* , 不可能使用代理，*Hibernate*会采取预先抓取！
❿	entity-name (可选): 被关联的类的实体名。

有两种不同的一对一关联：

- 主键关联
- 惟一外键关联

主键关联不需要额外的表字段；如果两行是通过这种一对一关系相关联的，那么这两行就共享同样的主关键字值。所以如果你希望两个对象通过主键一对一关联，你必须确认它们被赋予同样的标识值！

比如说，对下面的 Employee 和 Person 进行主键一对一关联:

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

现在我们必须确保PERSON和EMPLOYEE中相关的字段是相等的。我们使用一个被成为 `foreign` 的特殊的hibernate标识符生成策略：

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

一个刚刚保存的 `Person` 实例被赋予和该 `Person` 的 `employee` 属性所指向的 `Employee` 实例同样的关键字值。

另一种方式是一个外键和一个惟一关键字对应，上面的 `Employee` 和 `Person` 的例子，如果使用这种关联方式，可以表达成：

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique
```

如果在 `Person` 的映射加入下面几句，这种关联就是双向的：

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. 自然ID(natural-id)

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

我们建议使用代用键（键值不具备实际意义）作为主键，我们仍然应该尝试为所有的实体采用自然的键值作为（附加——译者注）标示。自然键（natural key）是单个或组合属性，他们必须唯一且非空。如果它还是不可变的那就更理想了。在 `<natural-id>` 元素中列出自然键的属性。Hibernate会帮你生成必须的唯一键值和非空约束，你的映射会更加的明显易懂（原文是self-documenting，自我注解）。

我们强烈建议你实现 `equals()` 和 `hashCode()` 方法,来比较实体的自然键属性。

这一映射不是为了把自然键作为主键而准备的。

- `mutable` (可选, 默认为 `false`): 默认情况下，自然标识属性被假定为不可变的（常量）。

5.1.13. 组件(component), 动态组件(dynamic-component)

`<component>` 元素把子对象的一些元素与父类对应的表的一些字段映射起来。 然后组件可以定义它们自己的属性、组件或者集合。参见后面的“Components”一章。

```
<component
        name="propertyName"
        class="className"
        insert="true|false"
        update="true|false"
        access="field|property|ClassName"
        lazy="true|false"
        optimistic-lock="true|false"
        unique="true|false"
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

❶	**name** : 属性名
❷	`class` (可选 - 默认为通过反射得到的属性类型):组件(子)类的名字。
❸	`insert` : 被映射的字段是否出现在SQL的 `INSERT` 语句中?
❹	`update` : 被映射的字段是否出现在SQL的 `UPDATE` 语句中?
❺	`access` (可选 - 默认是 `property`): Hibernate用来访问属性的策略。
❻	`lazy` (可选 - 默认是 `false`): 表明此组件应在实例变量第一次被访问的时候延迟加载(需要编译时字节码装置器)
❼	`optimistic-lock` (可选 - 默认是 `true`):表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号(Version)
❽	`unique` (可选 - 默认是 `false`):表明组件映射的所有字段上都有唯一性约束

其 `<property>` 子标签为子类的一些属性与表字段之间建立映射。

`<component>` 元素允许加入一个 `<parent>` 子元素，在组件类内部就可以有一个指向其容器的实体的反向引用。

<dynamic-component> 元素允许把一个 Map 映射为组件，其属性名对应map的键值。 参见第 8.5 节 “动态组件 （Dynamic components）”.

5.1.14. properties

<properties> 元素允许定义一个命名的逻辑分组(grouping)包含一个类中的多个属性。 这个元素最重要的用处是允许多个属性的组合作为 `property-ref` 的目标(target)。 这也是定义多字段唯一约束的一种方便途径。

```
<properties
        name="logicalName"
        insert="true|false"
        update="true|false"
        optimistic-lock="true|false"
        unique="true|false"
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

❶	**name**: 分组的逻辑名称 - 不是 实际属性的名称.
❷	`insert`: 被映射的字段是否出现在SQL的 `INSERT` 语句中?
❸	`update`: 被映射的字段是否出现在SQL的 `UPDATE` 语句中?
❹	`optimistic-lock` (可选 - 默认是 `true`):表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号(Version)
❺	`unique` (可选 - 默认是 `false`):表明组件映射的所有字段上都有唯一性约束

例如，如果我们有如下的 <properties> 映射:

```
<class name="Person">
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

然后，我们可能有一些遗留的数据关联，引用 `Person` 表的这个唯一键，而不是主键。

```
<many-to-one name="person"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

我们并不推荐这样使用，除非在映射遗留数据的情况下。

5.1.15. 子类(subclass)

最后，多态持久化需要为父类的每个子类都进行定义。对于“每一棵类继承树对应一个表”的策略来说，就需要使用 <subclass> 定义。

```
<subclass
        name="ClassName"
        discriminator-value="discriminator_value"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

•	**name** ：子类的全限定名。
•	discriminator-value(辨别标志) (可选 - 默认为类名):一个用于区分每个独立的子类的值。
•	proxy(代理) (可选): 指定一个类或者接口，在延迟装载时作为代理使用。
•	lazy (可选, 默认是 true): 设置为 lazy="false" 禁止使用延迟抓取

每个子类都应该定义它自己的持久化属性和子类。 <version> 和 <id> 属性可以从根父类继承下来。在一棵继承树上的每个子类都必须定义一个唯一的 discriminator-value 。如果没有指定，就会使用Java类的全限定名。

更多关于继承映射的信息, 参考 第 9 章 继承映射(*Inheritance Mappings*)章节.

5.1.16. 连接的子类(joined-subclass)

此外，每个子类可能被映射到他自己的表中(每个子类一个表的策略)。被继承的状态通过和超类的表关联得到。我们使用 <joined-subclass> 元素。

```
<joined-subclass
        name="ClassName"
        table="tablename"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

❶	**name**：子类的全限定名。
❷	table：子类的表名.
❸	proxy (可选): 指定一个类或者接口，在延迟装载时作为代理使用。
❹	lazy (可选, 默认是 true): 设置为 lazy="false" 禁止使用延迟装载。

这种映射策略不需要指定辨别标志(discriminator)字段。但是，每一个子类都必须使用 <key> 元素指定一个表字段来持有对象的标识符。本章开始的映射可以被用如下方式重写：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

更多关于继承映射的信息，参考第 9 章 继承映射*(Inheritance Mappings)*。

5.1.17. 联合子类(union-subclass)

第三种选择是仅仅映射类继承树中具体类部分到表中(每个具体类一张表的策略)。其中，每张表定义了类的所有持久化状态，包括继承的状态。在 Hibernate 中，并不需要完全显式地映射这样的继承树。你可以简单地使用单独的 `<class>` 定义映射每个类。然而，如果你想使用多态关联(例如，一个对类继承树中超类的关联)，你需要使用 `<union-subclass>` 映射。

```
<union-subclass
        name="ClassName"
        table="tablename"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

❶	**`name`** : 子类的全限定名。
❷	`table` : 子类的表名
❸	`proxy` (可选): 指定一个类或者接口，在延迟装载时作为代理使用。
❹	`lazy` (可选, 默认是 `true`): 设置为 `lazy="false"` 禁止使用延迟装载。

这种映射策略不需要指定辨别标志(discriminator)字段。

更多关于继承映射的信息，参考第 9 章 继承映射*(Inheritance Mappings)*。

5.1.18. 连接(join)

使用 `<join>` 元素，假若在表之间存在一对一关联,可以将一个类的属性映射到多张表中。

```
<join
        table="tablename"
        schema="owner"
        catalog="catalog"
        fetch="join|select"
        inverse="true|false"
        optional="true|false">

        <key ... />

        <property ... />
        ...
</join>
```

❶	**`table`**：被连接表的名称。
❷	`schema` (可选):覆盖由根 `<hibernate-mapping>` 元素指定的模式名称。
❸	`catalog` (可选): 覆盖由根 `<hibernate-mapping>` 元素指定的目录名称。
❹	`fetch` (可选 - 默认是 `join`): 如果设置为默认值 `join`， Hibernate 将使用一个内连接来得到这个类或其超类定义的 `<join>` ，而使用一个外连接来得到其子类定义的 `<join>` 。如果设置为 `select` ，则 Hibernate 将为子类定义的 `<join>` 使用顺序选择。这仅在一行数据表示一个子类的对象的时候才会发生。对这个类和其超类定义的 `<join>` ，依然会使用内连接得到。
❺	`inverse` (可选 - 默认是 `false`): 如果打开，Hibernate 不会插入或者更新此连接定义的属性。
❻	`optional` (可选 - 默认是 `false`): 如果打开，Hibernate 只会在此连接定义的属性非空时插入一行数据，并且总是使用一个外连接来得到这些属性。

例如，一个人(person)的地址(address)信息可以被映射到单独的表中(并保留所有属性的值类型语义)：

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

此特性常常对遗留数据模型有用，我们推荐表个数比类个数少，以及细粒度的领域模型。然而，在单独的继承树上切换继承映射策略是有用的，后面会解释这点。

5.1.19. 键(key)

我们目前已经见到过 `<key>` 元素多次了。 这个元素在父映射元素定义了对新表的连接，并且在被连接表中定义了一个外键引用原表的主键的情况下经常使用。

```
<key
        column="columnname"
        on-delete="noaction|cascade"
        property-ref="propertyName"
        not-null="true|false"
        update="true|false"
        unique="true|false"
/>
```

❶	**`column` (可选):** 外键字段的名称。也可以通过嵌套的 **`<column>`** 指定。
❷	`on-delete` (可选, 默认是 `noaction`): 表明外键关联是否打开数据库级别的级联删除。
❸	`property-ref` (可选): 表明外键引用的字段不是原表的主键(提供给遗留数据)。
❹	`not-null` (可选): 表明外键的字段不可为空(这意味着无论何时外键都是主键的一部分)。
❺	`update` (可选): 表明外键决不应该被更新(这意味着无论何时外键都是主键的一部分)。
❻	`unique` (可选): 表明外键应有唯一性约束 (这意味着无论何时外键都是主键的一部分)。

对那些看重删除性能的系统，我们推荐所有的键都应该定义为 `on-delete="cascade"`，这样 Hibernate 将使用数据库级的 `ON CASCADE DELETE` 约束，而不是多个 `DELETE` 语句。 注意，这个特性会绕过 Hibernate 通常对版本数据(versioned data)采用的乐观锁策略。

`not-null` 和 `update` 属性在映射单向一对多关联的时候有用。如果你映射一个单向一对多关联到非空的(non-nullable)外键，你必须用 `<key not-null="true">` 定义此键字段。

5.1.20. 字段和规则元素（column and formula elements）

任何接受 `column` 属性的映射元素都可以选择接受 <column> 子元素。同样的， `formula` 子元素也可以替换 <formula> 属性。

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

`column` 和 `formula` 属性甚至可以在同一个属性或关联映射中被合并来表达，例如，一些奇异的连接条件。

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

5.1.21. 引用(import)

假设你的应用程序有两个同样名字的持久化类，但是你不想在Hibernate查询中使用他们的全限定名。除了依赖 `auto-import="true"` 以外，类也可以被显式地“import(引用)”。你甚至可以引用没有被明确映射的类和接口。

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"
        rename="ShortName"
/>
```

❶	**`class`** : 任何**Java**类的全限定名。
❷	`rename` (可选 - 默认为类的全限定名): 在查询语句中可以使用的名字。

5.1.22. any

这是属性映射的又一种类型。 `<any>` 映射元素定义了一种从多个表到类的多态关联。这种类型的映射常常需要多于一个字段。第一个字段持有被关联实体的类型，其他的字段持有标识符。对这种类型的关联来说，不可能指定一个外键约束，所以这当然不是映射(多态)关联的通常的方式。你只应该在非常特殊的情况下使用它(比如，审计log，用户会话数据等等)。

`meta-type` 属性使得应用程序能指定一个将数据库字段的值映射到持久化类的自定义类型。这个持久化类包含有用 `id-type` 指定的标识符属性。 你必须指定从meta-type的值到类名的映射。

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"
        id-type="idtypename"
        meta-type="metatypename"
        cascade="cascade_style"
        access="field|property|ClassName"
        optimistic-lock="true|false"
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

❶	`name`: 属性名
❷	`id-type`: 标识符类型
❸	`meta-type` (可选 -默认是 `string`): 允许辨别标志(discriminator)映射的任何类型
❹	`cascade` (可选 -默认是`none`): 级联的类型
❺	`access` (可选 -默认是 `property`): Hibernate 用来访问属性值的策略。
❻	`optimistic-lock` (可选 -默认是 `true`): 表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号(Version)

5.2. Hibernate 的类型

5.2.1. 实体(Entities)和值(values)

为了理解很多与持久化服务相关的Java语言级对象的行为，我们需要把它们分为两类：

实体*entity* 独立于任何持有实体引用的对象。与通常的Java模型相比，不再被引用的对象会被当作垃圾收集掉。实体必须被显式的保存和删除(除非保存和删除是从父实体向子实体引发的级联)。这和ODMG模型中关于对象通过可触及保持持久性有一些不同——比较起来更加接近应用程序对象通常在一个大系统中的使用方法。实体支持循环引用和交叉引用，它们也可以加上版本信息。

一个实体的持久状态包含指向其他实体和值类型实例的引用。值可以是原始类型，集合(不是集合中的对象)，组件或者特定的不可变对象。与实体不同，值(特别是集合和组件)是通过可触及性来进行持久化和删除的。因为值对象(和原始类型数据)是随着包含他们的实体而被持久化和删除的，他们不能被独立的加上版本信息。值没有独立的标识，所以他们不能被两个实体或者集合共享。

直到现在，我们都一直使用术语“持久类”(persistent class)来代表实体。我们仍然会这么做。 然而严格说来，不是所有的用户自定义的，带有持久化状态的类都是实体。组件就是用户自定义类，却是值语义的。 `java.lang.String` 类型的java属性也是值语义的。给了这个定义以后，我们可以说所有JDK提供的类型(类)都是值类型的语义，而用于自定义类型可能被映射为实体类型或值类型语义。采用哪种类型的语义取决于开发人员。在领域模型中，寻找实体类的一个好线索是共享引用指向这个类的单一实例，而组合或聚合通常被转化为值类型。

我们会在本文档中重复碰到这两个概念。

挑战在于将java类型系统(和开发者定义的实体和值类型)映射到 SQL/数据库类型系统。Hibernate提供了连接两个系统之间的桥梁：对于实体类型，我们使用 `<class>` , `<subclass>` 等等。对于值类型，我们使用 `<property>` , `<component>` 及其他，通常跟随着 `type` 属性。这个属性的值是Hibernate 的映射类型的名字。Hibernate提供了许多现成的映射(标准的JDK值类型)。你也可以编写自己的映射类型并实现自定义的变换策略，随后我们会看到这点。

所有的Hibernate内建类型，除了collections以外，都支持空(null)语义。

5.2.2. 基本值类型

内建的 基本映射类型可以大致分为

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`

这些类型都对应Java的原始类型或者其封装类，来符合(特定厂商的)SQL 字段类型。 `boolean, yes_no` 和 `true_false` 都是Java 中 `boolean` 或者 `java.lang.Boolean` 的另外说法。

`string`

从 `java.lang.String` 到 `VARCHAR` (或者 Oracle的 `VARCHAR2`)的映射。

`date, time, timestamp`

从 `java.util.Date` 和其子类到SQL类型 `DATE` , `TIME` 和 `TIMESTAMP` (或等价类型)的映射。

`calendar, calendar_date`

从 `java.util.Calendar` 到SQL 类型 `TIMESTAMP` 和 `DATE` (或等价类型)的映射。

`big_decimal, big_integer`

从 `java.math.BigDecimal` 和 `java.math.BigInteger` 到 `NUMERIC` (或者 Oracle 的 `NUMBER` 类型)的映射。

`locale, timezone, currency`

从 `java.util.Locale` , `java.util.TimeZone` 和 `java.util.Currency` 到 `VARCHAR` (或者 Oracle 的 `VARCHAR2` 类型)的映射. `Locale` 和 `Currency` 的实例被映射为它们的ISO代码。 `TimeZone` 的实例被影射为它的 `ID` 。

`class`

从 `java.lang.Class` 到 `VARCHAR` (或者 Oracle 的 `VARCHAR2` 类型)的映射。 `Class` 被映射为它的全限定名。

`binary`

把字节数组(byte arrays)映射为对应的 SQL二进制类型。

`text`

把长Java字符串映射为SQL的 `CLOB` 或者 `TEXT` 类型。

`serializable`

把可序列化的Java类型映射到对应的SQL二进制类型。你也可以为一个并非默认为基本类型的可序列化Java类或者接口指定Hibernate类型 `serializable` 。

`clob, blob`

JDBC 类 `java.sql.Clob` 和 `java.sql.Blob` 的映射。某些程序可能不适合使用这个类型，因为blob和clob对象可能在一个事务之外是无法重用的。(而且, 驱动程序对这种类型的支持充满着补丁和前后矛盾。)

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date,`

一般来说，映射类型被假定为是可变的Java类型，只有对不可变Java类型，Hibernate会采取特定的优化措施，应用程序会把这些对象作为不可变对象处理。比如，你不应该对作为 `imm_timestamp` 映射的Date执行 `Date.setTime()` 。要改变属性的值，并且保存这一改变，应用程序必须对这一属性重新设置一个新的（不一样的）对象。

实体及其集合的唯一标识可以是除了 `binary` 、 `blob` 和 `clob` 之外的任何基础类型。(联合标识也是允许的，后面会说到。)

在 `org.hibernate.Hibernate` 中，定义了基础类型对应的 `Type` 常量。比如， `Hibernate.STRING` 代表 `string` 类型。

5.2.3. 自定义值类型

开发者创建属于他们自己的值类型也是很容易的。比如说，你可能希望持久化 `java.lang.BigInteger` 类型的属性，持久化成为 `VARCHAR` 字段。Hibernate没有内置这样一种类型。自定义类型能够映射一个属性(或集合元素)到不止一个数据库表字段。比如说，你可能有这样的Java属性： `getName()` / `setName()` ，这是 `java.lang.String` 类型的，对应的持久化到三个字段： `FIRST_NAME` ， `INITIAL` ， `SURNAME` 。

要实现一个自定义类型，可以实现 `org.hibernate.UserType` 或 `org.hibernate.CompositeUserType` 中的任一个，并且使用类型的Java全限定类名来定义属性。请查看 `org.hibernate.test.DoubleStringType` 这个例子，看看它是怎么做的。

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringTy
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

注意使用 `<column>` 标签来把一个属性映射到多个字段的做法。

`CompositeUserType` ， `EnhancedUserType` ， `UserCollectionType` ，和 `UserVersionType` 接口为更特殊的使用方式提供支持。

你甚至可以在一个映射文件中提供参数给一个 `UserType` 。 为了这样做，你的 `UserType` 必须实现 `org.hibernate.usertype.ParameterizedType` 接口。为了给自定义类型提供参数，你可以在映射文件中使用 `<type>` 元素。

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

现在， `UserType` 可以从传入的 `Properties` 对象中得到 `default` 参数的值。

如果你非常频繁地使用某一 `UserType` ，可以为他定义一个简称。这可以通过使用 `<typedef>` 元素来实现。Typedefs为一自定义类型赋予一个名称，并且如果此类型是参数化的，还可以包含一系列默认的参数值。

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" na
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

也可以根据具体案例通过属性映射中的类型参数覆盖在typedef中提供的参数。

尽管 Hibernate 内建的丰富的类型和对组件的支持意味着你可能很少 需要使用自定义类型。不过，为那些在你的应用中经常出现的(非实体)类使用自定义类型也是一个好方法。例如，一个 `MonetaryAmount` 类使用 `CompositeUserType` 来映射是不错的选择，虽然他可以很容易地被映射成组件。这样做的动机之一是抽象。使用自定义类型，以后假若你改变表示金额的方法时，它可以保证映射文件不需要修改。

5.3. 多次映射同一个类

对特定的持久化类，映射多次是允许的。这种情形下，你必须指定*entity name*来区别不同映射实体的对象实例。（默认情况下，实体名字和类名是相同的。）Hibernate在操作持久化对象、编写查询条件，或者把关联映射到指定实体时，允许你指定这个entity name（实体名字）。

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

注意这里关联是如何用 `entity-name` 来代替 `class` 的。

5.4. SQL中引号包围的标识符

你可通过在映射文档中使用反向引号
( )把表名或者字段名包围起来，以强制Hibernate在生成的SQL中把标识符用引号包围起来。
Dialect`(方言)来使用正确的引号风格(通常是双引号，但是在SQL Server中是括号，MySQL中是反向引号)。

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5. 其他元数据(Metadata)

XML 并不适用于所有人, 因此有其他定义Hibernate O/R 映射元数据(metadata)的方法。

5.5.1. 使用 XDoclet 标记

很多Hibernate使用者更喜欢使用XDoclet `@hibernate.tags` 将映射信息直接嵌入到源代码中。我们不会在本文档中涉及这个方法，因为严格说来，这属于XDoclet的一部分。然而，我们包含了如下使用XDoclet映射的 `Cat` 类的例子。

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /*
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
        this.mother = mother;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
```

```
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  inverse="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
```

```
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

参考Hibernate网站更多的Xdoclet和Hibernate的例子

5.5.2. 使用 JDK 5.0 的注解(Annotation)

JDK 5.0 在语言级别引入了 XDoclet 风格的标注，并且是类型安全的，在编译期进行检查。这一机制比XDoclet的注解更为强大，有更好的工具和IDE支持。例如，IntelliJ IDEA，支持JDK 5.0注解的自动完成和语法高亮 。EJB规范的新修订版(JSR-220)使用 JDK 5.0的注解作为entity beans的主要元数据(metadata)机制。Hibernate 3 实现了JSR-220 (the persistence API)的 `EntityManager` ，支持通过 *Hibernate Annotations*包定义映射元数据。这个包作为单独的部分下载，支持EJB3 (JSR-220)和Hibernate3的元数据。

这是一个被注解为EJB entity bean 的POJO类的例子

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;

    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order> orders;

    // Getter/setter and business methods
}
```

注意：对 JDK 5.0 注解 (和 JSR-220)支持的工作仍然在进行中,并未完成。更多细节请参阅Hibernate Annotations 模块。

5.6. 数据库生成属性（Generated Properties）

Generated properties指的是其值由数据库生成的属性。一般来说，如果对象有任何属性由数据库生成值，Hibernate应用程序需要进行 `刷新(refresh)` 。但如果把属性标明为generated，就可以转由Hibernate来负责这个动作。实际上。对定义了generated properties的实体,每当Hibernate执行一条SQL INSERT或者UPDATE语句，会立刻执行一条select来获得生成的值。

被标明为generated的属性还必须是 non-insertable和 non-updateable的。只有第 5.1.7 节 “版本（version）(可选)”，第 5.1.8 节 “timestamp (可选)”和第 5.1.9 节 “property”可以被标明为generated。

`never` (默认) 标明此属性值不是从数据库中生成。

`insert` - 标明此属性值在insert的时候生成，但是不会在随后的update时重新生成。比如说创建日期就归属于这类。注意虽然第 5.1.7 节 “版本（version）(可选)”和第 5.1.8 节 “timestamp (可选)”属性可以被标注为generated，但是不适用这个选项...

`always` - 标明此属性值在insert和update时都会被生成。

5.7. 辅助数据库对象(Auxiliary Database Objects)

Allows CREATE and DROP of arbitrary database objects, in conjunction with Hibernate's schema evolution tools, to provide the ability to fully define a user schema within the Hibernate mapping files. Although designed specifically for creating and dropping things like triggers or stored procedures, really any SQL command that can be run via a `java.sql.Statement.execute()` method is valid here (ALTERs, INSERTS, etc). There are essentially two modes for defining auxiliary database objects... 帮助CREATE和DROP任意数据库对象，与Hibernate的schema交互工具组合起来，可以提供在Hibernate映射文件中完全定义用户schema的能力。虽然这是为创建和销毁trigger(触发器）或stored procedure(存储过程）等特别设计的，实际上任何可以在 `java.sql.Statement.execute()` 方法中执行的SQL命令都可以在此使用（比如ALTER, INSERT，等等）。本质上有两种模式来定义辅助数据库对象...

第一种模式是在映射文件中显式声明CREATE和DROP命令：

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

第二种模式是提供一个类，这个类知道如何组织CREATE和DROP命令。这个特别类必须实现 `org.hibernate.mapping.AuxiliaryDatabaseObject` 接口。

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

还有，这些数据库对象可以特别指定为仅在特定的方言中才使用。

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/
        <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
    </database-object>
</hibernate-mapping>
```

第 6 章 集合类(Collections)映射

目录

6.1. 持久化集合类(Persistent collections)

(译者注：在阅读本章的时候，以后整个手册的阅读过程中，我们都会面临一个名词方面的问题，那就是“集合”。"Collections"和"Set"在中文里对应都被翻译为“集合”，但是他们的含义很不一样。Collections是一个超集，Set是其中的一种。大部分情况下，本译稿中泛指的未加英文注明的“集合”，都应当理解为“Collections”。在有些二者同时出现，可能造成混淆的地方，我们用“集合类”来特指“Collecions”,“集合(Set)”来指"Set"，一般都会在后面的括号中给出英文。希望大家在阅读时联系上下文理解，不要造成误解。 与此同时，“元素”一词对应的英文“element”，也有两个不同的含义。其一为集合的元素，是内存中的一个变量；另一含义则是XML文档中的一个标签所代表的元素。也请注意区别。 本章中,特别是后半部分是需要反复阅读才能理解清楚的。如果遇到任何疑问,请记住,英文版本的reference是惟一标准的参考资料。)

Hibernate要求持久化集合值字段必须声明为接口，比如：

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

实际的接口可能是 `java.util.Set` , `java.util.Collection` , `java.util.List` , `java.util.Map` , `java.util.SortedSet` , `java.util.SortedMap` 或者...任何你喜欢的类型！("任何你喜欢的类型" 代表你需要编写 `org.hibernate.usertype.UserCollectionType` 的实现.)

注意我们是如何用一个 `HashSet` 实例来初始化实例变量的.这是用于初始化新创建(尚未持久化)的类实例中集合值属性的最佳方法。当你持久化这个实例时——比如通过调用 `persist()` ——Hibernate 会自动把 `HashSet` 替换为Hibernate自己的 `Set` 实现。观察下面的错误：

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); //Okay, kittens collection is a Set
(HashSet) cat.getKittens(); //Error!
```

根据不同的接口类型，被Hibernate注射的持久化集合类的表现类似 `HashMap`, `HashSet`, `TreeMap`, `TreeSet` or `ArrayList`。

集合类实例具有值类型的通常行为。当被持久化对象引用后，他们会自动被持久化，当不再被引用后，自动被删除。假若实例被从一个持久化对象传递到另一个，它的元素可能从一个表转移到另一个表。两个实体不能共享同一个集合类实例的引用。因为底层关系数据库模型的原因，集合值属性无法支持空值语义；Hibernate对空的集合引用和空集合不加区别。

你不需要过多的为此担心。就如同你平时使用普通的Java集合类一样来使用持久化集合类。只是要确认你理解了双向关联的语义（后文讨论）。

6.2. 集合映射（ Collection mappings ）

用于映射集合类的Hibernate映射元素取决于接口的类型。比如， <set> 元素用来映射 Set 类型的属性。

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

除了 <set> ,还有 <list> , <map> , <bag> , <array> 和 <primitive-array> 映射元素。 <map> 具有代表性：

```
<map
    name="propertyName"
    table="table_name"
    schema="schema_name"
    lazy="true|extra|false"
    inverse="true|false"
    cascade="all|none|save-update|delete|all-delete-orphan|delete-
    sort="unsorted|natural|comparatorClass"
    order-by="column_name asc|desc"
    where="arbitrary sql where condition"
    fetch="join|select|subselect"
    batch-size="N"
    access="field|property|ClassName"
    optimistic-lock="true|false"
    mutable="true|false"
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

❶	`name` 集合属性的名称
❷	`table` （可选——默认为属性的名称）这个集合表的名称(不能在一对多的关联关系中使用)
❸	`schema` (可选) 表的schema的名称, 他将覆盖在根元素中定义的schema
❹	`lazy` (可选--默认为true) 可以用来关闭延迟加载(false)，指定一直使用预先抓取,或者打开"extra-lazy" 抓取，此时大多数操作不会初始化集合类(适用于非常大的集合)
❺	`inverse` (可选——默认为 `false`) 标记这个集合作为双向关联关系中的方向一端。
❻	`cascade` (可选——默认为 `none`) 让操作级联到子实体
❼	`sort` (可选)指定集合的排序顺序, 其可以为自然的(`natural`)或者给定一个用来比较的类。
❽	`order-by` (可选, 仅用于jdk1.4) 指定表的字段(一个或几个)再加上asc或者desc(可选), 定义Map,Set和Bag的迭代顺序
❾	`where` (可选) 指定任意的SQL where条件, 该条件将在重新载入或者删除这个集合时使用(当集合中的数据仅仅是所有可用数据的一个子集时这个条件非常有用)
❿	`fetch` (可选, 默认为 `select`) 用于在外连接抓取、通过后续select抓取和通过后续subselect抓取之间选择。
⓫	`batch-size` (可选, 默认为 `1`) 指定通过延迟加载取得集合实例的批处理块大小（"batch size"）。
⓬	`access` (可选-默认为属性property):Hibernate取得集合属性值时使用的策略
⓭	`乐观锁` (可选 - 默认为 `true`): 对集合的状态的改变会是否导致其所属的实体的版本增长。 (对一对多关联来说，关闭这个属性常常是有理的)
⓮	`mutable(可变)` (可选 - 默认为 `true`): 若值为 `false` ,表明集合中的元素不会改变（在某些情况下可以进行一些小的性能优化）。

6.2.1. 集合外键(Collection foreign keys)

集合实例在数据库中依靠持有集合的实体的外键加以辨别。此外键作为集合关键字段（*collection key column*）（或多个字段）加以引用。集合关键字段通过 <key> 元素映射。

在外键字段上可能具有非空约束。对于大多数集合来说，这是隐含的。对单向一对多关联来说，外键字段默认是可以为空的，因此你可能需要指明 not-null="true" 。

```
<key column="productSerialNumber" not-null="true"/>
```

外键约束可以使用 ON DELETE CASCADE 。

```
<key column="productSerialNumber" on-delete="cascade"/>
```

对 <key> 元素的完整定义，请参阅前面的章节。

6.2.2. 集合元素（Collection elements）

集合几乎可以包含任何其他的Hibernate类型，包括所有的基本类型、自定义类型、组件，当然还有对其他实体的引用。存在一个重要的区别：位于集合中的对象可能是根据“值”语义来操作（其声明周期完全依赖于集合持有者），或者它可能是指向另一个实体的引用，具有其自己的生命周期。在后者的情况下，被作为集合持有的状态考虑的，只有两个对象之间的“连接”。

被包容的类型被称为集合元素类型（*collection element type*）。集合元素通过 <element> 或 <composite-element> 映射，或在其是实体引用的时候，通过 <one-to-many> 或 <many-to-many> 映射。前两种用于使用值语义映射元素，后两种用于映射实体关联。

6.2.3. 索引集合类(Indexed collections)

所有的集合映射，除了set和bag语义的以外，都需要指定一个集合表的索引字段(*index column*)——用于对应到数组索引，或者 `List` 的索引，或者 `Map` 的关键字。通过 <map-key> , `Map` 的索引可以是任何基础类型；若通过 <map-key-many-to-many> ，它也可以是一个实体引用；若通过 <composite-map-key> ，它还可以是一个组合类型。数组或列表的索引必须是 `integer` 类型，并且使用 <list-index> 元素定义映射。被映射的字段包含有顺序排列的整数（默认从0开始）。

```
<map-key
        column="column_name"
        formula="any SQL expression"
        type="type_name"
        node="@attribute-name"
        length="N"/>
```

❶	**`column`** **(可选):**保存集合索引值的字段名。
❷	`formula` (可选): 用于计算map关键字的SQL公式
❸	`type` (必须):映射键(map key)的类型。

```
<map-key-many-to-many
        column="column_name"
        formula="any SQL expression"
        class="ClassName"
/>
```

❶	**`column`** **(可选):**集合索引值中外键字段的名称
❷	`formula` (可选): 用于计算map关键字的外键的SQL公式
❸	`class` (必需):映射的键(map key)使用的实体类。

假若你的表没有一个索引字段,当你仍然希望使用 `List` 作为属性类型,你应该把此属性映射为Hibernate <*bag*>。从数据库中获取的时候，bag不维护其顺序，但也可选择性的进行排序。

从集合类可以产生很大一部分映射，覆盖了很多常见的关系模型。我们建议你试验schema生成工具，来体会一下不同的映射声明是如何被翻译为数据库表的。

6.2.4. 值集合于多对多关联(Collections of values and many-to-many associations)

任何值集合或者多对多关联需要专用的具有一个或多个外键字段的*collection table*、一个或多个*collection element column*，以及还可能有一个或多个索引字段。

对于一个值集合, 我们使用 <element> 标签。

```
<element
        column="column_name"
        formula="any SQL expression"
        type="typename"
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

❶	**`column` (可选):**保存集合元素值的字段名。
❷	`formula` (可选): 用于计算元素的SQL公式
❸	`type` (必需):集合元素的类型

多对多关联*(many-to-many association)* 使用 <many-to-many> 元素定义.

```
<many-to-many
        column="column_name"
        formula="any SQL expression"
        class="ClassName"
        fetch="select|join"
        unique="true|false"
        not-found="ignore|exception"
        entity-name="EntityName"
        property-ref="propertyNameFromAssociatedClass"
        node="element-name"
        embed-xml="true|false"
    />
```

❶	**`column` (可选):** 这个元素的外键关键字段名
❷	`formula` (可选): 用于计算元素外键值的SQL公式.
❸	`class` (必需): 关联类的名称
❸	`outer-join` (可选 - 默认为 `auto`): 在Hibernate系统参数中 `hibernate.use_outer_join` 被打开的情况下,该参数用来允许使用outer join来载入此集合的数据。
❹	为此关联打开外连接抓取或者后续select抓取。这是特殊情况；对于一个实体及其指向其他实体的多对多关联进全预先抓取（使用一条单独的 `SELECT` ），你不仅需要对集合自身打开 `join` ，也需要对 `<many-to-many>` 这个内嵌元素打开此属性。
❺	对外键字段允许DDL生成的时候生成一个惟一约束。这使关联变成了一个高效的一对多关联。（此句存疑：原文为This makes the association multiplicity effectively one to many.)
❻	`not-found` (可选 - 默认为 `exception`): 指明引用的外键中缺少某些行该如何处理： `ignore` 会把缺失的行作为一个空引用处理。
❼	`entity-name` (可选): 被关联的类的实体名，作为 `class` 的替代。
❽	`property-ref` ：(可选) 被关联到此外键(foreign key)的类中的对应属性的名字。若未指定，使用被关联类的主键。

例子：首先, 一组字符串：

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

包含一组整数的bag(还设置了 `order-by` 参数指定了迭代的顺序)：

```
<bag name="sizes"
        table="item_sizes"
        order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

一个实体数组,在这个案例中是一个多对多的关联(注意这里的实体是自动管理生命周期的对象（lifecycle objects），`cascade="all"`):

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
    <key column="personId"/>
    <list-index column="sortOrder"/>
    <many-to-many column="addressId" class="Address"/>
</array>
```

一个map,通过字符串的索引来指明日期：

```
<map name="holidays"
        table="holidays"
        schema="dbo"
        order-by="hol_name asc">
    <key column="id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

一个组件的列表：（下一章讨论）

```
<list name="carComponents"
        table="CarComponents">
    <key column="carId"/>
    <list-index column="sortOrder"/>
    <composite-element class="CarComponent">
        <property name="price"/>
        <property name="type"/>
        <property name="serialNumber" column="serialNum"/>
    </composite-element>
</list>
```

6.2.5. 一对多关联（One-to-many Associations）

一对多关联__通过外键连接两个类对应的表,而没有中间集合表。 这个关系模型失去了一些Java集合的语义:

- 一个被包含的实体的实例只能被包含在一个集合的实例中
- 一个被包含的实体的实例只能对应于集合索引的一个值中

一个从 `Product` 到 `Part` 的关联需要关键字字段,可能还有一个索引字段指向 `Part` 所对应的表。 `<one-to-many>` 标记指明了一个一对多的关联。

```
<one-to-many
        class="ClassName"
        not-found="ignore|exception"
        entity-name="EntityName"
        node="element-name"
        embed-xml="true|false"
    />
```

❶	**`class` (必须):**被关联类的名称。
❷	`not-found` (可选 - 默认为 `exception`): 指明若缓存的标示值关联的行缺失,该如何处理: `ignore` 会把缺失的行作为一个空关联处理。
❸	`entity-name` (可选): 被关联的类的实体名，作为 `class` 的替代。

例子

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="org.hibernate.Bar"/>
</set>
```

注意: `<one-to-many>` 元素不需要定义任何字段。 也不需要指定表名。

重要提示:如果 `一对多` 关联中的外键字段定义成 `NOT NULL` ,你必须把 `<key>` 映射声明为 `not-null="true"` ,或者使用双向关联，并且标明 `inverse="true"` 。参阅本章后面关于双向关联的讨论。

下面的例子展示一个 `Part` 实体的map,把name作为关键字。(`partName` 是 `Part` 的持久化属性)。注意其中的基于公式的索引的用法。

```
<map name="parts"
        cascade="all">
    <key column="productId" not-null="true"/>
    <map-key formula="partName"/>
    <one-to-many class="Part"/>
</map>
```

6.3. 高级集合映射（Advanced collection mappings）

6.3.1. 有序集合（Sorted collections）

Hibernate支持实现 `java.util.SortedMap` 和 `java.util.SortedSet` 的集合。你必须在映射文件中指定一个比较器：

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

`sort` 属性中允许的值包括 `unsorted`，`natural` 和某个实现了 `java.util.Comparator` 的类的名称。

分类集合的行为事实上象 `java.util.TreeSet` 或者 `java.util.TreeMap` 。

如果你希望数据库自己对集合元素排序，可以利用 `set`，`bag` 或者 `map` 映射中的 `order-by` 属性。这个解决方案只能在jdk1.4或者更高的jdk版本中才可以实现(通过LinkedHashSet或者 LinkedHashMap实现)。 它是在SQL查询中完成排序，而不是在内存中。

```
<set name="aliases" table="person_aliases" order-by="lower(name) as
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

注意: 这个 `order-by` 属性的值是一个SQL排序子句而不是HQL的！

关联还可以在运行时使用集合 `filter()` 根据任意的条件来排序。

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name
```

6.3.2. 双向关联（Bidirectional associations）

双向关联允许通过关联的任一端访问另外一端。在Hibernate中, 支持两种类型的双向关联:

一对多（one-to-many）

Set或者bag值在一端, 单独值(非集合)在另外一端

多对多（many-to-many）

两端都是set或bag值

要建立一个双向的多对多关联，只需要映射两个many-to-many关联到同一个数据库表中，并再定义其中的一端为*inverse*(使用哪一端要根据你的选择，但它不能是一个索引集合)。

这里有一个many-to-many的双向关联的例子;每一个category都可以有很多items,每一个items可以属于很多categories：

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="CATEGORY_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

如果只对关联的反向端进行了改变，这个改变不会被持久化。 这表示Hibernate为每个双向关联在内存中存在两次表现,一个从A连接到B,另一个从B连接到A。如果你回想一下Java对象模型，我们是如何在Java中创建多对多关系的，这可以让你更容易理解：

```
category.getItems().add(item);          // The category now "knows'
item.getCategories().add(category);     // The item now "knows" abo

session.persist(item);                   // The relationship won''t
session.persist(category);               // The relationship will b
```

非反向端用于把内存中的表示保存到数据库中。

要建立一个一对多的双向关联，你可以通过把一个一对多关联，作为一个多对一关联映射到到同一张表的字段上，并且在"多"的那一端定义 `inverse="true"` 。

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

在“一”这一端定义 `inverse="true"` 不会影响级联操作，二者是正交的概念！

6.3.3. 双向关联，涉及有序集合类

对于有一端是 <list> 或者 <map> 的双向关联，需要加以特别考虑。假若子类中的一个属性映射到索引字段，没问题，我们仍然可以在集合类映射上使用 `inverse="true"` ：

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

但是，假若子类中没有这样的属性存在，我们不能认为这个关联是真正的双向关联（信息不对称，在关联的一端有一些另外一端没有的信息）。在这种情况下，我们不能使用 `inverse="true"` 。我们需要这样用：

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

注意在这个映射中，关联中集合类"值"一端负责来更新外键.TODO: Does this really result in some unnecessary update statements?

6.3.4. 三重关联（Ternary associations）

有三种可能的途径来映射一个三重关联。第一种是使用一个 `Map` ，把一个关联作为其索引：

```
<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections">
    <key column="incoming_node_id"/>
    <map-key-many-to-many column="outgoing_node_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

第二种方法是简单的把关联重新建模为一个实体类。这使我们最经常使用的方法。

最后一种选择是使用复合元素，我们会在后面讨论

6.3.5. 使用`<idbag>`

如果你完全信奉我们对于“联合主键（composite keys）是个坏东西”，和“实体应该使用（无机的）自己生成的代用标识符（surrogate keys）”的观点，也许你会感到有一些奇怪，我们目前为止展示的多对多关联和值集合都是映射成为带有联合主键的表的！现在，这一点非常值得争辩；看上去一个单纯的关联表并不能从代用标识符中获得什么好处（虽然使用组合值的集合可能会获得一点好处）。不过，Hibernate提供了一个（一点点试验性质的）功能，让你把多对多关联和值集合应得到一个使用代用标识符的表去。

`<idbag>` 属性让你使用bag语义来映射一个 `List` (或 `Collection`)。

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

你可以理解， `<idbag>` 人工的id生成器，就好像是实体类一样！集合的每一行都有一个不同的人造关键字。但是，Hibernate没有提供任何机制来让你取得某个特定行的人造关键字。

注意 `<idbag>` 的更新性能要比普通的 `<bag>` 高得多！Hibernate可以有效的定位到不同的行，分别进行更新或删除工作，就如同处理一个list, map或者set一样。

在目前的实现中，还不支持使用 `identity` 标识符生成器策略来生成 `<idbag>` 集合的标识符。

6.4. 集合例子（Collection example）

在前面的几个章节的确非常令人迷惑。 因此让我们来看一个例子。这个类：

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=childrer

    ....
    ....
}
```

这个类有一个 `Child` 的实例集合。如果每一个子实例至多有一个父实例, 那么最自然的映射是一个one-to-many的关联关系：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

在以下的表定义中反应了这个映射关系：

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(2
alter table child add constraint childfk0 (parent_id) references pa
```

如果父亲是必须的, 那么就可以使用双向one-to-many的关联了：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id
    </class>

</hibernate-mapping>
```

请注意 `NOT NULL` 的约束:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references pa
```

另外，如果你绝对坚持这个关联应该是单向的，你可以对 `<key>` 映射声明 `NOT NULL` 约束：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

另外一方面,如果一个子实例可能有多个父实例, 那么就应该使用many-to-many关联：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

表定义：

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(2
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) referen
alter table childset add constraint childsetfk1 (child_id) referenc
```

更多的例子,以及一个完整的父/子关系映射的排练,请参阅第 21 章 示例：父子关系 *(Parent Child Relationships)*.

甚至可能出现更加复杂的关联映射,我们会在下一章中列出所有可能性。

第 7 章 关联关系映射

目录

7.1. 介绍

关联关系映射通常情况是最难配置正确的。在这个部分中，我们从单向关系映射开始，然后考虑双向关系映射，由浅至深讲述一遍典型的案例。在所有的例子中，我们都使用 `Person` 和 `Address` 。

我们根据映射关系是否涉及连接表以及多样性来划分关联类型。

在传统的数据建模中，允许为Null值的外键被认为是一种不好的实践，因此我们所有的例子中都使用不允许为Null的外键。这并不是Hibernate的要求，即使你删除掉不允许为Null的约束，Hibernate映射一样可以工作的很好。

7.2. 单向关联（Unidirectional associations）

7.2.1. 多对一(many to one)

单向*many-to-one*关联是最常见的单向关联关系。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, address
create table Address ( addressId bigint not null primary key )
```

7.2.2. 一对一（one to one）

基于外键关联的单向一对一关联和单向多对一关联几乎是一样的。唯一的不同就是单向一对一关联中的外键字段具有唯一性约束。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, address
create table Address ( addressId bigint not null primary key )
```

基于主键关联的单向一对一关联通常使用一个特定的id生成器。（请注意，在这个例子中我们掉换了关联的方向。）

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.2.3. 一对多（one to many）

基于外键关联的单向一对多关联是一种很少见的情况，并不推荐使用。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, perso
```

我们认为对于这种关联关系最好使用连接表。

7.3. 使用连接表的单向关联（Unidirectional associations with join tables）

7.3.1. 一对多(one to many)

基于连接表的单向一对多关联 应该优先被采用。请注意，通过指定 `unique="true"` ，我们可以把多样性从多对多改变为一对多。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint nc
create table Address ( addressId bigint not null primary key )
```

7.3.2. 多对一（many to one）

基于连接表的单向多对一关联在关联关系可选的情况下应用也很普遍。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key,
create table Address ( addressId bigint not null primary key )
```

7.3.3. 一对一（one to one）

基于连接表的单向一对一关联非常少见，但也是可行的。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key,
create table Address ( addressId bigint not null primary key )
```

7.3.4. 多对多（many to many）

最后，还有 单向多对多关联.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bi
create table Address ( addressId bigint not null primary key )
```

7.4. 双向关联（Bidirectional associations）

7.4.1. 一对多（one to many) / 多对一（many to one）

双向多对一关联 是最常见的关联关系。（这也是标准的父/子关联关系。）

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, address
create table Address ( addressId bigint not null primary key )
```

如果你使用 `List` (或者其他有序集合类)，你需要设置外键对应的 `key` 列为 `not null` ,让Hibernate来从集合端管理关联，维护每个元素的索引（通过设置 `update="false"` and `insert="false"` 来对另一端反向操作）。

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class>
```

假若集合映射的 <key> 元素对应的底层外键字段是 NOT NULL 的，那么为这一key元素定义 not-null="true" 是很重要的。不要仅仅为可能的嵌套 <column> 元素定义 not-null="true" ， <key> 元素也是需要的。

7.4.2. 一对一（one to one）

基于外键关联的双向一对一关联也很常见。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, address
create table Address ( addressId bigint not null primary key )
```

基于主键关联的一对一关联需要使用特定的id生成器。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.5. 使用连接表的双向关联（Bidirectional associations with join tables）

7.5.1. 一对多（one to many） /多对一（ many to one）

基于连接表的双向一对多关联。注意 `inverse="true"` 可以出现在关联的任意一端，即collection端或者join端。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bi
create table Address ( addressId bigint not null primary key )
```

7.5.2. 一对一（one to one）

基于连接表的双向一对一关联极为罕见，但也是可行的。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key,
create table Address ( addressId bigint not null primary key )
```

7.5.3. 多对多（many to many）

最后，还有 双向多对多关联.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"  table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true"  table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bi
create table Address ( addressId bigint not null primary key )
```

7.6. 更复杂的关联映射

更复杂的关联连接极为罕见。 通过在映射文档中嵌入SQL片断，Hibernate也可以处理更为复杂的情况。比如，假若包含历史帐户数据的表定义了 `accountNumber`, `effectiveEndDate` 和 `effectiveStartDate` 字段，按照下面映射：

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula>case when effectiveEndDate is null then 1 else 0 e
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

那么我们可以对目前*(current)*实例(其 `effectiveEndDate` 为null)使用这样的关联映射:

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula>'1'</formula>
</many-to-one>
```

更复杂的例子,假想 `Employee` 和 `Organization` 之间的关联是通过一个 `Employment` 中间表维护的,而中间表中填充了很多历史雇员数据。那“雇员的最新雇主”这个关联（最新雇主就是 `startDate` 最后的那个）可以这样映射：

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join>
```

使用这一功能时可以充满创意，但通常更加实用的是用HQL或条件查询来处理这些情形。

第 8 章 组件（Component）映射

目录

组件(Component)这个概念在Hibernate中几处不同的地方为了不同的目的被重复使用.

8.1. 依赖对象（Dependent objects）

组件(Component)是一个被包含的对象，在持久化的过程中，它被当作值类型，而并非一个实体的引用。在这篇文档中，组件这一术语指的是面向对象的合成概念（而并不是系统构架层次上的组件的概念）。举个例子, 你对人(Person)这个概念可以像下面这样来建模：

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

在持久化的过程中, `姓名(Name)` 可以作为 `人(Person)` 的一个组件。需要注意的是:你应该为 `姓名` 的持久化属性定义getter和setter方法,但是你不需要实现任何的接口或申明标识符字段。

以下是这个例子的Hibernate映射文件:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute op
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

人员(Person)表中将包括 `pid` , `birthday` , `initial` , `first` 和 `last` 等字段。

就像所有的值类型一样, 组件不支持共享引用。 换句话说, 两个人可能重名, 但是两个Person对象应该包含两个独立的Name对象, 只不过这两个Name对象具有“同样”的值。 组件的值可以为空, 其定义如下。 每当Hibernate重新加载一个包含组件

的对象,如果该组件的所有字段为空，Hibernate将假定整个组件为空。 在大多数情况下,这样假定应该是没有问题的。

组件的属性可以是任意一种Hibernate类型（包括集合, 多对多关联， 以及其它组件等等）。嵌套组件不应该被当作一种特殊的应用(Nested components should not be considered an exotic usage)。 Hibernate倾向于支持细致的(fine-grained)对象模型。

<component> 元素还允许有 <parent> 子元素，用来表明component类中的一个属性是指向包含它的实体的引用。

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date">
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the Per
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component&gt;
</class>
```

8.2. 在集合中出现的依赖对象 (Collections of dependent objects)

Hibernate支持组件的集合(例如: 一个元素是姓名(Name)这种类型的数组)。 你可以使用 `<composite-element>` 标签替代 `<element>` 标签来定义你的组件集合。

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/&gt;
    <composite-element class="eg.Name"> <!-- class attribute requi
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>;
    </composite-element>
</set>
```

注意，如果你定义的Set包含组合元素(composite-element)， 正确地实现 `equals()` 和 `hashCode()` 是非常重要的。

组合元素可以包含组件，但是不能包含集合。如果你的组合元素自身包含组件, 你必须使用 `<nested-composite-element>` 标签。这是一个相当特殊的案例 - 在一个组件的集合里，那些组件本身又可以包含其他的组件。这个时候你就应该考虑一下使用one-to-many关联是否会更恰当。 尝试对这个组合元素重新建模为一个实体－但是需要注意的是，虽然Java模型和重新建模前是一样的，关系模型和持久性语义会有细微的变化。

请注意如果你使用 `<set>` 标签,一个组合元素的映射不支持可能为空的属性. 当删除对象时， Hibernate必须使用每一个字段的值来确定一条记录(在组合元素表中，没有单独的关键字段)， 如果有为null的字段，这样做就不可能了。你必须作出一个选择，要么在组合元素中使用不能为空的属性，要么选择使用 `<list>` , `<map>` , `<bag>` 或者 `<idbag>` 而不是 `<set>` 。

组合元素有个特别的用法是它可以包含一个 `<many-to-one>` 元素。类似这样的映射允许你将一个many-to-many关联表映射为组合元素的集合。(A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class.) 接下来的的例子是从 `Order` 到 `Item` 的一个多对多的关联关系, 关联属性是 `purchaseDate` , `price` 和 `quantity` 。

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class a
        </composite-element>
    </set>
</class>
```

当然，当你定义Item时，你无法引用这些purchase，因此你无法实现双向关联查询。记住组件是值类型，并且不允许共享引用。某一个特定的 `Purchase` 可以放在 `Order` 的集合中，但它不能同时被 `Item` 所引用。

其实组合元素的这个用法可以扩展到三重或多重关联:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails" class="eg.Purchase'
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

在查询中，表达组合元素的语法和关联到其他实体的语法是一样的。

8.3. 组件作为Map的索引（Components as Map indices ）

`<composite-map-key>` 元素允许你映射一个组件类作为一个 `Map` 的 key，前提是你必须正确的在这个类中重写了 `hashCode()` 和 `equals()` 方法。

8.4. 组件作为联合标识符(Components as composite identifiers)

你可以使用一个组件作为一个实体类的标识符。 你的组件类必须满足以下要求：

- 它必须实现 java.io.Serializable 接口
- 它必须重新实现 equals() 和 hashCode() 方法, 始终和组合关键字在数据库中的概念保持一致

注意：在*Hibernate3*中，第二个要求并非是*Hibernate*强制必须的。但最好这样做。

你不能使用一个 IdentifierGenerator 产生组合关键字。一个应用程序必须分配它自己的标识符。

使用 <composite-id> 标签(并且内嵌 <key-property> 元素)代替通常的 <id> 标签。比如, OrderLine 类具有一个主键，这个主键依赖于 Order 的(联合)主键。

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class>
```

现在，任何指向 OrderLine 的外键都是复合的。在你的映射文件中，必须为其他类也这样声明。例如，一个指向 OrderLine 的关联可能被这样映射：

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one>
```

（注意在各个地方 `<column>` 标签都是 `column` 属性的替代写法。）

指向 `OrderLine` 的 `多对多` 关联也使用联合外键:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set>
```

在 `Order` 中, `OrderLine` 的集合则是这样:

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set>
```

(与通常一样, `<one-to-many>` 元素不声明任何列.)

假若 `OrderLine` 本身拥有一个集合,它也具有组合外键。

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key>   <!-- a collection inherits the composite key type -
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
```

8.5. 动态组件 （Dynamic components）

你甚至可以映射 `Map` 类型的属性：

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
```

从 `<dynamic-component>` 映射的语义上来讲，它和 `<component>` 是相同的。 这种映射类型的优点在于通过修改映射文件，就可以具有在部署时检测真实属性的能力。利用一个DOM解析器，也可以在程序运行时操作映射文件。 更好的是，你可以通过 `Configuration` 对象来访问（或者修改）Hibernate的运行时元模型。

第 9 章 继承映射(Inheritance Mappings)

目录

9.1. 三种策略

Hibernate支持三种基本的继承映射策略：

- 每个类分层结构一张表(table per class hierarchy)
- 每个子类一张表(table per subclass)
- 每个具体类一张表(table per concrete class)

此外，Hibernate还支持第四种稍有不同的多态映射策略：

- 隐式多态(implicit polymorphism)

对于同一个继承层次内的不同分支，可以采用不同的映射策略，然后用隐式多 态来完成跨越整个层次的多态。但是在同一个 <class> 根元素 下，Hibernate 不支持混合了元素 <subclass> 、 <joined-subclass> 和 <union-subclass> 的映射。在同一个 <class> 元素下，可以混合使用 “每个类分层结构一张表”（table per hierarchy） 和“每个子类一张表”（table per subclass） 这两种映射策略，这是通过结合元素 <subclass> 和 <join> 来实现的（见后）。

在多个映射文件中，可以直接在 hibernate-mapping 根下定义 subclass ， union-subclass 和 joined-subclass 。也就是说，你可以仅加入一个新的映射文件来扩展类层次。你必须在subclass的映射中指明 extends 属性，给出一个之前定义的超类的名字。注意，在以前，这一功能对映射文件的顺序有严格的要求，从Hibernate 3开始，使用extends关键字的时侯，对映射文件的顺序不再有要求；但在每个映射文件里，超类必须在子类之前定义。

```
<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-value
         <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>
```

9.1.1. 每个类分层结构一张表(Table per class hierarchy)

假设我们有接口 `Payment` 和它的几个实现类： `CreditCardPayment` , `CashPayment` , 和 `ChequePayment` 。则“每个类分层结构一张表”(Table per class hierarchy)的映射代码如下所示：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT'
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

采用这种策略只需要一张表即可。它有一个很大的限制：要求那些由子类定义的字段， 如 `CCTYPE` ， 不能有 `非空(NOT NULL)` 约束。

9.1.2. 每个子类一张表(Table per subclass)

对于上例中的几个类而言，采用“每个子类一张表”的映射策略，代码如下所示：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

需要四张表。三个子类表通过主键关联到超类表(因而关系模型实际上是一对一关联)。

9.1.3. 每个子类一张表(Table per subclass)，使用辨别标志(Discriminator)

注意，对“每个子类一张表”的映射策略，Hibernate的实现不需要辨别字段，而其他的对象/关系映射工具使用了一种不同于Hibernate的实现方法，该方法要求在超类表中有一个类型辨别字段(type discriminator column)。Hibernate采用的方法更 难实现，但从关系（数据库）的角度来看，按理说它更正确。若你愿意使用带有辨别字 段的“每个子类一张表”的策略，你可以结合使用 `<subclass>` 与 `<join>` ，如下所示：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT"
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

可选的声明 `fetch="select"` ，是用来告诉Hibernate，在查询超类时， 不要使用外部连接(outer join)来抓取子类 `ChequePayment` 的数据。

9.1.4. 混合使用“每个类分层结构一张表”和“每个子类一张表”

你甚至可以采取如下方法混和使用“每个类分层结构一张表”和“每个子类一张表”这两种策略：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT"
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

对上述任何一种映射策略而言，指向根类 `Payment` 的 关联是使用 <many-to-one> 进行映射的。

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

9.1.5. 每个具体类一张表(Table per concrete class)

对于“每个具体类一张表”的映射策略，可以采用两种方法。第一种方法是使用 `<union-subclass>` 。

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT"
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

这里涉及三张与子类相关的表。每张表为对应类的所有属性（包括从超类继承的属性）定义相应字段。

这种方式的局限在于，如果一个属性在超类中做了映射，其字段名必须与所有子类表中定义的相同。(我们可能会在Hibernate的后续发布版本中放宽此限制。) 不允许在联合子类(union subclass)的继承层次中使用标识生成器策略(identity generator strategy), 实际上, 主键的种子(primary key seed)不得不为同一继承层次中的全部被联合子类所共用.

假若超类是抽象类，请使用 `abstract="true"` 。当然，假若它不是抽象的，需要一个额外的表（上面的例子中，默认是 `PAYMENT` ），来保存超类的实例。

9.1.6. Table per concrete class, using implicit polymorphism

另一种可供选择的方法是采用隐式多态：

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

注意，我们没有在任何地方明确的提及接口 Payment 。同时注意 Payment 的属性在每个子类中都进行了映射。如果你想避免重复， 可以考虑使用XML实体(例如：位于 DOCTYPE 声明内的 [<!ENTITY allproperties SYSTEM "allproperties.xml">] 和映射中的 &allproperties;)。

这种方法的缺陷在于，在Hibernate执行多态查询时(polymorphic queries)无法生成带 UNION 的SQL语句。

对于这种映射策略而言，通常用 <any> 来实现到 Payment 的多态关联映射。

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

9.1.7. 隐式多态和其他继承映射混合使用

对这一映射还有一点需要注意。因为每个子类都在各自独立的元素 <class> 中映射(并且 Payment 只是一个接口)，每个子类可以很容易的成为另一 个继承体系中的一部分！(你仍然可以对接口 Payment 使用多态查询。)

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

我们还是没有明确的提到 Payment 。 如果我们针对接口 Payment 执行查询 —— 如 from Payment —— Hibernate 自动返回 CreditCardPayment (和它的子类，因为 它们也实现了接口 Payment)、 CashPayment 和 Chequepayment 的实例， 但不返回 NonelectronicTransaction 的实例。

9.2. 限制

对“每个具体类映射一张表”（table per concrete-class）的映射策略而言，隐式多态的 方式有一定的限制。而 <union-subclass> 映射的限制则没有那 么严格。

下面表格中列出了在Hibernte中“每个具体类一张表”的策略和隐式多态的限制。

表 **9.1.** 继承映射特性**(Features of inheritance mappings)**

继承策略 **(Inheritance strategy)**	多态多对一	多态一对一	多
每个类分层结构一张表	<many-to-one>	<one-to-one>	<or
每个子类一张表	<many-to-one>	<one-to-one>	<or
每个具体类一张表(union-subclass)	<many-to-one>	<one-to-one>	<or (仅对于 inve 情况)
每个具体类一张表(隐式多态)	<any>	不支持	不支持

第 10 章 与对象共事

目录

Hibernate是完整的对象/关系映射解决方案，它提供了对象状态管理*(state management)*的功能，使开发者不再需要理会底层数据库系统的细节。 也就是说，相对于常见的JDBC/SQL持久层方案中需要 `管理SQL语句` ，Hibernate采用了更自然的面向对象的视角来持久化Java应用中的数据。

换句话说，使用Hibernate的开发者应该总是关注对象的状态*(state)*，不必考虑SQL语句的执行。 这部分细节已经由Hibernate掌管妥当，只有开发者在进行系统性能调优的时候才需要进行了解。

10.1. Hibernate对象状态(object states)

Hibernate定义并支持下列对象状态(state):

- 瞬时*(Transient)* - 由 `new` 操作符创建，且尚未与Hibernate `Session` 关联的对象被认定为瞬时(Transient)的。瞬时(Transient)对象不会被持久化到数据库中，也不会被赋予持久化标识(identifier)。 如果瞬时(Transient)对象在程序中没有被引用，它会被垃圾回收器(garbage collector)销毁。 使用Hibernate `Session` 可以将其变为持久(Persistent)状态。(Hibernate会自动执行必要的SQL语句)
- 持久*(Persistent)* - 持久(Persistent)的实例在数据库中有对应的记录，并拥有一个持久化标识(identifier)。 持久(Persistent)的实例可能是刚被保存的，或刚被加载的，无论哪一种，按定义，它存在于相关联的 `Session` 作用范围内。 Hibernate会检测到处于持久(Persistent)状态的对象的任何改动，在当前操作单元(unit of work)执行完毕时将对象数据(state)与数据库同步(synchronize)。 开发者不需要手动执行 `UPDATE` 。将对象从持久(Persistent)状态变成瞬时(Transient)状态同样也不需要手动执行 `DELETE` 语句。
- 脱管*(Detached)* - 与持久(Persistent)对象关联的 `Session` 被关闭后，对象就变为脱管(Detached)的。 对脱管(Detached)对象的引用依然有效，对象可继续被修改。脱管(Detached)对象如果重新关联到某个新的 `Session` 上， 会再次转变为持久(Persistent)的(在Detached其间的改动将被持久化到数据库)。 这个功能使得一种编程模型，即中间会给用户思考时间(user think-time)的长时间运行的操作单元(unit of work)的编程模型成为可能。 我们称之为应用程序事务，即从用户观点看是一个操作单元(unit of work)。

接下来我们来细致的讨论下状态(states)及状态间的转换(state transitions)（以及触发状态转换的Hibernate方法）。

10.2. 使对象持久化

Hibernate认为持久化类(persistent class)新实例化的对象是瞬时*(Transient)*的。 我们可通过将瞬时(Transient)对象与session关联而把它变为持久*(Persistent)*的。

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

如果 `Cat` 的持久化标识(identifier)是 `generated` 类型的， 那么该标识(identifier)会自动在 `save()` 被调用时产生并分配给 `cat` 。 如果 `Cat` 的持久化标识(identifier)是 `assigned` 类型的，或是一个复合主键(composite key)， 那么该标识(identifier)应当在调用 `save()` 之前手动赋予给 `cat` 。 你也可以按照EJB3 early draft中定义的语义，使用 `persist()` 替代 `save()` 。

此外，你可以用一个重载版本的 `save()` 方法。

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

如果你持久化的对象有关联的对象(associated objects)（例如上例中的 `kittens` 集合） 那么对这些对象（译注：pk和kittens）进行持久化的顺序是任意的（也就是说可以先对kittens进行持久化也可以先对pk进行持久化）， 除非你在外键列上有 `NOT NULL` 约束。 Hibernate不会违反外键约束，但是如果你用错误的顺序持久化对象（译注：在pk持久化之前持久化kitten），那么可能会违反 `NOT NULL` 约束。

通常你不会为这些细节烦心，因为你很可能会使用Hibernate的 传播性持久化*(transitive persistence)*功能自动保存相关联那些对象。 这样连违反 `NOT NULL` 约束的情况都不会出现了 - Hibernate会管好所有的事情。 传播性持久化(transitive persistence)将在本章稍后讨论。

10.3. 装载对象

如果你知道某个实例的持久化标识(identifier)，你就可以使用 `Session` 的 `load()` 方法来获取它。 `load()` 的另一个参数是指定类的.class对象。 本方法会创建指定类的持久化实例，并从数据库加载其数据(state)。

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Lo
```

此外, 你可以把数据(state)加载到指定的对象实例上（覆盖掉该实例原来的数据）。

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

请注意如果没有匹配的数据库记录， `load()` 方法可能抛出无法恢复的异常(unrecoverable exception)。 如果类的映射使用了代理(proxy)， `load()` 方法会返回一个未初始化的代理，直到你调用该代理的某方法时才会去访问数据库。 若你希望在某对象中创建一个指向另一个对象的关联，又不想在从数据库中装载该对象时同时装载相关联的那个对象，那么这种操作方式就用得上的了。 如果为相应类映射关系设置了 `batch-size` ， 那么使用这种操作方式允许多个对象被一批装载（因为返回的是代理，无需从数据库中抓取所有对象的数据）。

如果你不确定是否有匹配的行存在，应该使用 `get()` 方法，它会立刻访问数据库，如果没有对应的记录，会返回null。

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

你甚至可以选用某个 `LockMode` ，用SQL的 `SELECT ... FOR UPDATE` 装载对象。 请查阅API文档以获取更多信息。

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

注意，任何关联的对象或者包含的集合都不会被以 `FOR UPDATE` 方式返回， 除非你指定了 `lock` 或者 `all` 作为关联(association)的级联风格(cascade style)。

任何时候都可以使用 `refresh()` 方法强迫装载对象和它的集合。如果你使用数据库触发器功能来处理对象的某些属性，这个方法就很有用了。

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

此处通常会出现一个重要问题: Hibernate会从数据库中装载多少东西？会执行多少条相应的SQL `SELECT` 语句？ 这取决于抓取策略*(fetching strategy)*，会在第 19.1 节 “ 抓取策略(Fetching strategies) ”中解释。

10.4. 查询

如果不知道所要寻找的对象的持久化标识，那么你需要使用查询。Hibernate支持强大且易于使用的面向对象查询语言(HQL)。 如果希望通过编程的方式创建查询，Hibernate提供了完善的按条件(Query By Criteria, QBC)以及按样例(Query By Example, QBE)进行查询的功能。 你也可以用原生SQL(native SQL)描述查询，Hibernate额外提供了将结果集(result set)转化为对象的支持。

10.4.1. 执行查询

HQL和原生SQL(native SQL)查询要通过为 `org.hibernate.Query` 的实例来表达。 这个接口提供了参数绑定、结果集处理以及运行实际查询的方法。 你总是可以通过当前 `Session` 获取一个 `Query` 对象：

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kitten
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

一个查询通常在调用 `list()` 时被执行，执行结果会完全装载进内存中的一个集合(collection)。 查询返回的对象处于持久(persistent)状态。如果你知道的查询只会返回一个对象，可使用 `list()` 的快捷方式 `uniqueResult()` 。 注意，使用集合预先抓取的查询往往会返回多次根对象（他们的集合类都被初始化了）。你可以通过一个集合来过滤这些重复对象。

10.4.1.1. 迭代式获取结果(Iterating results)

某些情况下，你可以使用 `iterate()` 方法得到更好的性能。 这通常是你预期返回的结果在session，或二级缓存(second-level cache)中已经存在时的情况。 如若不然， `iterate()` 会比 `list()` 慢，而且可能简单查询也需要进行多次数据库访问： `iterate()` 会首先使用*1*条语句得到所有对象的持久化标识(identifiers)，再根据持久化标识执行*n*条附加的select语句实例化实际的对象。

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeline
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

10.4.1.2. 返回元组(tuples)的查询

（译注：元组(tuples)指一条结果行包含多个对象） Hibernate查询有时返回元组(tuples)，每个元组(tuples)以数组的形式返回:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.moth
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten  = tuple[0];
    Cat mother  = tuple[1];
    ....
}
```

10.4.1.3. 标量(Scalar)结果

查询可在 `select` 从句中指定类的属性，甚至可以调用SQL统计(aggregate)函数。 属性或统计结果被认定为"标量(Scalar)"的结果（而不是持久(persistent state)的实体）。

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

10.4.1.4. 绑定参数

接口 `Query` 提供了对命名参数(named parameters)、JDBC风格的 `问号(?)` 参数进行绑定的方法。 不同于*JDBC*，*Hibernate*对参数从*0*开始计数。 命名参数(named parameters)在查询字符串中是形如 `:name` 的标识符。 命名参数(named parameters)的优点是:

- 命名参数(named parameters)与其在查询串中出现的顺序无关
- 它们可在同一查询串中多次出现
- 它们本身是自我说明的

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = 
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = 
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in
q.setParameterList("namesList", names);
List cats = q.list();
```

10.4.1.5. 分页

如果你需要指定结果集的范围（希望返回的最大行数/或开始的行数），应该使用 `Query` 接口提供的方法:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate 知道如何将这个有限定条件的查询转换成你的数据库的原生SQL(native SQL)。

10.4.1.6. 可滚动遍历(Scrollable iteration)

如果你的JDBC驱动支持可滚动的 `ResuleSet` ， `Query` 接口可以使用 `ScrollableResults` ，允许你在查询结果中灵活游走。

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat c
                            "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cat

}
cats.close()
```

请注意，使用此功能需要保持数据库连接（以及游标(cursor)）处于一直打开状态。 如果你需要断开连接使用分页功能，请使用 `setMaxResult()` / `setFirstResult()`

10.4.1.7. 外置命名查询(Externalizing named queries)

你可以在映射文件中定义命名查询(named queries)。（如果你的查询串中包含可能被解释为XML标记(markup)的字符，别忘了用 `CDATA` 包裹起来。）

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

参数绑定及执行以编程方式(programatically)完成：

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

请注意实际的程序代码与所用的查询语言无关，你也可在元数据中定义原生SQL(native SQL)查询， 或将原有的其他的查询语句放在配置文件中，这样就可以让Hibernate统一管理，达到迁移的目的。

也请注意在 `<hibernate-mapping>` 元素中声明的查询必须有一个全局唯一的名字,而在 `<class>` 元素中声明的查询自动具有全局名,是通过类的全名加以限定的。比如 `eg.Cat.ByNameAndMaximumWeight` 。

10.4.2. 过滤集合

集合过滤器*(filter)*是一种用于一个持久化集合或者数组的特殊的查询。查询字符串中可以使用 `"this"` 来引用集合中的当前元素。

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.clas
    .list()
);
```

返回的集合可以被认为是一个包(bag, 无顺序可重复的集合(collection))，它是所给集合的副本。 原来的集合不会被改动（这与“过滤器(filter)”的隐含的含义不符，不过与我们期待的行为一致）。

请注意过滤器(filter)并不需要 `from` 子句（当然需要的话它们也可以加上）。过滤器(filter)不限定于只能返回集合元素本身。

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

即使无条件的过滤器(filter)也是有意义的。例如，用于加载一个大集合的子集：

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

10.4.3. 条件查询(Criteria queries)

HQL极为强大，但是有些人希望能够动态的使用一种面向对象API创建查询，而非在他们的Java代码中嵌入字符串。对于那部分人来说，Hibernate提供了直观的 `Criteria` 查询API。

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

`Criteria` 以及相关的 `样例(Example)` API将会再第 15 章 条件查询(Criteria Queries) 中详细讨论。

10.4.4. 使用原生SQL的查询

你可以使用 `createSQLQuery()` 方法，用SQL来描述查询，并由Hibernate将结果集转换成对象。 请注意，你可以在任何时候调用 `session.connection()` 来获得并使用JDBC `Connection` 对象。 如果你选择使用Hibernate的API, 你必须把SQL别名用大括号包围起来:

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class]
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

和Hibernate查询一样，SQL查询也可以包含命名参数和占位参数。 可以在第 16 章 *Native SQL查询*找到更多关于Hibernate中原生SQL(native SQL)的信息。

10.5. 修改持久对象

事务中的持久实例（就是通过 `session` 装载、保存、创建或者查询出的对象） 被应用程序操作所造成的任何修改都会在 `Session` 被刷出（*flushed*）的时候被持久化（本章后面会详细讨论）。 这里不需要调用某个特定的方法（比如 `update()`，设计它的目的是不同的）将你的修改持久化。 所以最直接的更新一个对象的方法就是在 `Session` 处于打开状态时 `load()` 它，然后直接修改即可：

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69)
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and pe
```

有时这种程序模型效率低下，因为它在同一Session里需要一条SQL `SELECT` 语句（用于加载对象） 以及一条SQL `UPDATE` 语句(持久化更新的状态)。 为此 Hibernate提供了另一种途径，使用脱管(detached)实例。

请注意*Hibernate*本身不提供直接执行 *`UPDATE`* 或 *`DELETE`* 语句的*API*。 *Hibernate*提供的是__状态管理*(state management)*服务，你不必考虑要使用的语句*(statements)*。 JDBC是出色的执行SQL语句的API，任何时候调用 `session.connection()` 你都可以得到一个JDBC `Connection` 对象。 此外，在联机事务处理(OLTP)程序中，大量操作(mass operations)与对象/关系映射的观点是相冲突的。 Hibernate的将来版本可能会提供专门的进行大量操作(mass operation)的功能。 参考第 13 章 批量处理（*Batch processing*），寻找一些可用的批量(batch)操作技巧。

10.6. 修改脱管(Detached)对象

很多程序需要在某个事务中获取对象，然后将对象发送到界面层去操作，最后在一个新的事务保存所做的修改。 在高并发访问的环境中使用这种方式，通常使用附带版本信息的数据来保证这些“长“工作单元之间的隔离。

Hibernate通过提供 `Session.update()` 或 `Session.merge()` 重新关联脱管实例的办法来支持这种模型。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

如果具有 `catId` 持久化标识的 `Cat` 之前已经被 `另一Session(secondSession)` 装载了， 应用程序进行重关联操作(reattach)的时候会抛出一个异常。

如果你确定当前session没有包含与之具有相同持久化标识的持久实例，使用 `update()` 。 如果想随时合并你的的改动而不考虑session的状态，使用 `merge()` 。 换句话说，在一个新session中通常第一个调用的是 `update()` 方法，以便保证重新关联脱管(detached)对象的操作首先被执行。

如果希望相关联的脱管对象（通过引用“可到达”的脱管对象）的数据也要更新到数据库时（并且也仅仅在这种情况）， 可以对该相关联的脱管对象单独调用 `update()` 当然这些可以自动完成，即通过使用传播性持久化*(transitive persistence)*，请看第 10.11 节 “传播性持久化(transitive persistence)”。

`lock()` 方法也允许程序重新关联某个对象到一个新session上。不过，该脱管(detached)的对象必须是没有修改过的！

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate
sess.lock(pk, LockMode.UPGRADE);
```

请注意， `lock()` 可以搭配多种 `LockMode` ， 更多信息请阅读API文档以及关于事务处理(transaction handling)的章节。重新关联不是 `lock()` 的唯一用途。

其他用于长时间工作单元的模型会在第 11.3 节 “乐观并发控制(Optimistic concurrency control)”中讨论。

10.7. 自动状态检测

Hibernate的用户曾要求一个既可自动分配新持久化标识(identifier)保存瞬时(transient)对象，又可更新/重新关联脱管(detached)实例的通用方法。`saveOrUpdate()` 方法实现了这个功能。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat ha
secondSession.saveOrUpdate(mate);  // save the new instance (mate h
```

`saveOrUpdate()` 用途和语义可能会使新用户感到迷惑。 首先，只要你没有尝试在某个session中使用来自另一session的实例，你就应该不需要使用 `update()`， `saveOrUpdate()`，或 `merge()`。有些程序从来不用这些方法。

通常下面的场景会使用 `update()` 或 `saveOrUpdate()`：

- 程序在第一个session中加载对象
- 该对象被传递到表现层
- 对象发生了一些改动
- 该对象被返回到业务逻辑层
- 程序调用第二个session的 `update()` 方法持久这些改动

`saveOrUpdate()` 做下面的事:

- 如果对象已经在本session中持久化了，不做任何事
- 如果另一个与本session关联的对象拥有相同的持久化标识(identifier)，抛出一个异常
- 如果对象没有持久化标识(identifier)属性，对其调用 `save()`
- 如果对象的持久标识(identifier)表明其是一个新实例化的对象，对其调用 `save()`

- 如果对象是附带版本信息的（通过 <version> 或 <timestamp> ） 并且版本属性的值表明其是一个新实例化的对象， `save()` 它。
- 否则 `update()` 这个对象

`merge()` 可非常不同:

- 如果session中存在相同持久化标识(identifier)的实例，用用户给出的对象的状态覆盖旧有的持久实例
- 如果session没有相应的持久实例，则尝试从数据库中加载，或创建新的持久化实例
- 最后返回该持久实例
- 用户给出的这个对象没有被关联到session上，它依旧是脱管的

10.8. 删除持久对象

使用 `Session.delete()` 会把对象的状态从数据库中移除。 当然，你的应用程序可能仍然持有一个指向已删除对象的引用。所以，最好这样理解： `delete()` 的用途是把一个持久实例变成瞬时(transient)实例。

```
sess.delete(cat);
```

你可以用你喜欢的任何顺序删除对象，不用担心外键约束冲突。当然，如果你搞错了顺序，还是有可能引发在外键字段定义的 `NOT NULL` 约束冲突。 例如你删除了父对象，但是忘记删除孩子们。

10.9. 在两个不同数据库间复制对象

偶尔会用到不重新生成持久化标识(identifier)，将持久实例以及其关联的实例持久到不同的数据库中的操作。

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

`ReplicationMode`决定在和数据库中已存在记录由冲突时， `replicate()` 如何处理。

- `ReplicationMode.IGNORE` - 忽略它
- `ReplicationMode.OVERWRITE` - 覆盖相同的行
- `ReplicationMode.EXCEPTION` - 抛出异常
- `ReplicationMode.LATEST_VERSION` - 如果当前的版本较新，则覆盖，否则忽略

这个功能的用途包括使录入的数据在不同数据库中一致，产品升级时升级系统配置信息，回滚non-ACID事务中的修改等等。（译注，non-ACID，非ACID;ACID，Atomic，Consistent，Isolated and Durable的缩写）

10.10. Session刷出(flush)

每间隔一段时间，`Session` 会执行一些必需的SQL语句来把内存中的对象的状态同步到JDBC连接中。这个过程被称为刷出(*flush*)，默认会在下面的时间点执行：

- 在某些查询执行之前
- 在调用 `org.hibernate.Transaction.commit()` 的时候
- 在调用 `Session.flush()` 的时候

涉及的SQL语句会按照下面的顺序发出执行：

1. 所有对实体进行插入的语句，其顺序按照对象执行 `Session.save()` 的时间顺序
2. 所有对实体进行更新的语句
3. 所有进行集合删除的语句
4. 所有对集合元素进行删除，更新或者插入的语句
5. 所有进行集合插入的语句
6. 所有对实体进行删除的语句，其顺序按照对象执行 `Session.delete()` 的时间顺序

（有一个例外是，如果对象使用 `native` 方式来生成ID（持久化标识）的话，它们一执行save就会被插入。）

除非你明确地发出了 `flush()` 指令，关于Session何时会执行这些JDBC调用是完全无法保证的，只能保证它们执行的前后顺序。 当然，Hibernate保证， `Query.list(..)` 绝对不会返回已经失效的数据，也不会返回错误数据。

也可以改变默认的设置，来让刷出(flush)操作发生的不那么频繁。 `FlushMode` 类定义了三种不同的方式。 仅在提交时刷出(仅当Hibernate的 `Transaction` API被使用时有效)， 按照刚才说的方式刷出， 以及除非明确使用 `flush()` 否则从不刷出。 最后一种模式对于那些需要长时间保持 `Session` 为打开或者断线状态的长时间运行的工作单元很有用。 (参见 第 11.3.2 节 “扩展周期的session和自动版本化”).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return sta

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

刷出(flush)期间，可能会抛出异常。（例如一个DML操作违反了约束）异常处理涉及到对Hibernate事务性行为的理解，因此我们将在第 11 章 事务和并发中讨论。

10.11. 传播性持久化(transitive persistence)

对每一个对象都要执行保存，删除或重关联操作让人感觉有点麻烦，尤其是在处理许多彼此关联的对象的时候。 一个常见的例子是父子关系。考虑下面的例子:

如果一个父子关系中的子对象是值类型(value typed)（例如，地址或字符串的集合）的，他们的生命周期会依赖于父对象，可以享受方便的级联操作(Cascading)，不需要额外的动作。 父对象被保存时，这些值类型(value typed)子对象也将被保存；父对象被删除时，子对象也将被删除。 这对将一个子对象从集合中移除是同样有效：Hibernate会检测到，并且因为值类型(value typed)的对象不可能被其他对象引用，所以Hibernate会在数据库中删除这个子对象。

现在考虑同样的场景，不过父子对象都是实体(entities)类型，而非值类型(value typed)（例如，类别与个体，或母猫和小猫）。 实体有自己的生命期，允许共享对其的引用（因此从集合中移除一个实体，不意味着它可以被删除）， 并且实体到其他关联实体之间默认没有级联操作的设置。 Hibernate默认不实现所谓的可到达即持久化（*persistence by reachability*）的策略。

每个Hibernate session的基本操作 - 包括 `persist(), merge(), saveOrUpdate(), delete(), lock(), refresh(), evi` - 都有对应的级联风格(cascade style)。 这些级联风格(cascade style)风格分别命名为 `create, merge, save-update, delete, lock, refresh, evict, replicate` 。 如果你希望一个操作被顺着关联关系级联传播，你必须在映射文件中指出这一点。例如：

```
<one-to-one name="person" cascade="persist"/>
```

级联风格(cascade style)是可组合的:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

你可以使用 `cascade="all"` 来指定全部操作都顺着关联关系级联(cascaded)。 默认值是 `cascade="none"` ，即任何操作都不会被级联(cascaded)。

注意有一个特殊的级联风格(cascade style) `delete-orphan` ，只应用于one-to-many关联，表明 `delete()` 操作 应该被应用于所有从关联中删除的对象。

建议:

- 通常在 <many-to-one> 或 <many-to-many> 关系中应用级联(cascade)没什么意义。 级联(cascade)通常在 <one-to-one> 和 <one-to-many> 关系中比较有用。

- 如果子对象的寿命限定在父亲对象的寿命之内，可通过指定 `cascade="all,delete-orphan"` 将其变为自动生命周期管理的对象 *(lifecycle object)*。
- 其他情况，你可根本不需要级联(cascade)。但是如果你认为你会经常在某个事务中同时用到父对象与子对象，并且你希望少打点儿字，可以考虑使用 `cascade="persist,merge,save-update"`。

可以使用 `cascade="all"` 将一个关联关系（无论是对值对象的关联，或者对一个集合的关联）标记为父/子关系的关联。 这样对父对象进行save/update/delete操作就会导致子对象也进行save/update/delete操作。

此外，一个持久的父对象对子对象的浅引用(mere reference)会导致子对象被同步save/update。 不过，这个隐喻(metaphor)的说法并不完整。除非关联是 `<one-to-many>` 关联并且被标记为 `cascade="delete-orphan"`， 否则父对象失去对某个子对象的引用不会导致该子对象被自动删除。 父子关系的级联(cascading)操作准确语义如下：

- 如果父对象被 `persist()`，那么所有子对象也会被 `persist()`
- 如果父对象被 `merge()`，那么所有子对象也会被 `merge()`
- 如果父对象被 `save()`， `update()` 或 `saveOrUpdate()`，那么所有子对象则会被 `saveOrUpdate()`
- 如果某个持久的父对象引用了瞬时(transient)或者脱管(detached)的子对象，那么子对象将会被 `saveOrUpdate()`
- 如果父对象被删除，那么所有子对象也会被 `delete()`
- 除非被标记为 `cascade="delete-orphan"` （删除“孤儿”模式，此时不被任何一个父对象引用的子对象会被删除）， 否则子对象失掉父对象对其的引用时，什么事也不会发生。 如果有特殊需要，应用程序可通过显式调用delete()删除子对象。

最后，注意操作的级联可能是在调用期*(call time)*或者写入期*(flush time)*作用到对象图上的。所有的操作，如果允许，都在操作被执行的时候级联到可触及的关联实体上。然而， `save-upate` 和 `delete-orphan` 是在 `Session` flush的时候才作用到所有可触及的被关联对象上的。

10.12. 使用元数据

Hibernate中有一个非常丰富的元级别(meta-level)的模型，含有所有的实体和值类型数据的元数据。 有时这个模型对应用程序本身也会非常有用。 比如说，应用程序可能在实现一种“智能”的深度拷贝算法时， 通过使用Hibernate的元数据来了解哪些对象应该被拷贝（比如，可变的值类型数据）， 那些不应该（不可变的值类型数据，也许还有某些被关联的实体）。

Hibernate提供了 `ClassMetadata` 接口， `CollectionMetadata` 接口和 `Type` 层次体系来访问元数据。 可以通过 `SessionFactory` 获取元数据接口的实例。

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associa
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCo
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

第 11 章 事务和并发

目录

Hibernate的事务和并发控制很容易掌握。Hibernate直接使用JDBC连接和JTA资源，不添加任何附加锁定 行为。我们强烈推荐你花点时间了解JDBC编程，ANSI SQL查询语言和你使用 的数据库系统的事务隔离规范。

Hibernate不锁定内存中的对象。你的应用程序会按照你的数据库事务的隔离级别规定的那样运作。幸亏有了 `Session` ，使得Hibernate通过标识符查找，和实体查询（不是返回标量值的报表查询）提供了可重复的读取（Repeatable reads）功能， `Session` 同时也是事务范围内的缓存（cache）。

除了对自动乐观并发控制提供版本管理，针对行级悲观锁定，Hibernate也提供了辅助的（较小的)API，它使用了 `SELECT FOR UPDATE` 的SQL语法。本章后面会讨论乐观并发控制和这个API。

我们从 `Configuration` 层、 `SessionFactory` 层, 和 `Session` 层开始讨论Hibernate的并行控制、数据库事务和应用 程序的长事务。

11.1. Session和事务范围(transaction scope)

`SessionFactory` 对象的创建代价很昂贵，它是线程安全的对象，它为所有的应用程序线程所共享。它只创建一次，通常是在应用程序启动的时候，由一个 `Configuraion` 的实例来创建。

`Session` 对象的创建代价比较小，是非线程安全的，对于单个请求，单个会话、单个的 工作单元而言，它只被使用一次，然后就丢弃。只有在需要的时候，一个 `Session` 对象 才会获取一个JDBC的 `Connection` （或一个 `Datasource` ）对象，因此假若不使用的时候它不消费任何资源。

此外我们还要考虑数据库事务。数据库事务应该尽可能的短，降低数据库中的锁争用。 数据库长事务会阻止你的应用程序扩展到高的并发负载。因此，假若在用户思考期间让数据库事务开着，直到整个工作单元完成才关闭这个事务，这绝不是一个好的设计。

一个操作单元(Unit of work)的范围是多大？单个的Hibernate `Session` 能跨越多个数据库事务吗？还是一个 `Session` 的作用范围对应一个数据库事务的范围？应该何时打开 `Session` ，何时关闭 `Session` ？，你又如何划分数据库事务的边界呢？

11.1.1. 操作单元(Unit of work)

首先，别用*session-per-operation*这种反模式了，也就是说，在单个线程中，不要因为一次简单的数据库调用，就打开和关闭一次 `Session` ！数据库事务也是如此。应用程序中的数据库调用是按照计划好的次序，分组为原子的操作单元。（注意，这也意味着，应用程 序中，在单个的SQL语句发送之后，自动事务提交(auto-commit)模式失效了。这种模式专门为SQL控制台操作设计的。 Hibernate禁止立即自动事务提交模式，或者期望应用服务器禁止立即自动事务提交模式。）数据库事务绝不是可有可无的，任何与数据库之间的通讯都必须在某个事务中进行，不管你是在读还是在写数据。对读数据而言，应该避免auto-commit行为，因为很多小的事务比一个清晰定义的工作单元性能差。后者也更容易维护和扩展。

在多用户的client/server应用程序中，最常用的模式是 每个请求一个会话*(session-per-request)*。在这种模式下，来自客户端的请求被发送到服务器端（即Hibernate持久化层运行的地方），一 个新的Hibernate `Session` 被打开，并且执行这个操作单元中所有的数据库操作。一旦操作完成（同时对客户端的响应也准备就绪），session被同步，然后关闭。你也可以使用单 个数据库事务来处理客户端请求，在你打开 `Session` 之后启动事务，在你关闭 `Session` 之前提交事务。会话和请求之间的关系是一对一的关系，这种模式对 于大多数应用程序来说是很棒的。

实现才是真正的挑战。Hibernate内置了对"当前session(current session)" 的管理，用于简化此模式。你要做的一切就是在服务器端要处理请求的时候，开启事务，在响应发送给客户之前结束事务。你可以用任何方式来完成这一操作，通常的方案有 `ServletFilter` ，在service方法中进行pointcut的AOP拦截器，或者proxy/interception容器。EJB容器是实现横切诸如EJB session bean上的事务分界，用CMT对事务进行声明等方面的标准手段。假若你决定使用编程式的事务分界，请参考本章后面讲到的Hibernate `Transaction` API，这对易用性和代码可移植性都有好处。

在任何时间，任何地方，你的应用代码可以通过简单的调用 `sessionFactory.getCurrentSession()` 来访问"当前session"，用于处理请求。你总是会得到当前数据库事务范围内的 `Session` 。在使用本地资源或JTA环境时，必须配置它，请参见第 2.5 节 “上下文相关的（Contextual）Session”。

有时，将 `Session` 和数据库事务的边界延伸到"展示层被渲染后"会带来便利。有些serlvet应用程序在对请求进行处理后，有个单独的渲染期，这种延伸对这种程序特别有用。假若你实现你自己的拦截器，把事务边界延伸到展示层渲染结束后非常容易。然而，假若你依赖有容器管理事务的EJB，这就不太容易了，因为事务会在EJB方法返回后结束，而那是在任何展示层渲染开始之前。请访问Hibernate网站和论坛，你可以找到*Open Session in View*这一模式的提示和示例。

11.1.2. 长对话

session-per-request模式不仅仅是一个可以用来设计操作单元的有用概念。很多业务处理都需 要一系列完整的与用户之间的交互，而这些用户是指对数据库有交叉访问的用户。在基于web的应用和企业 应用中，跨用户交互的数据库事务是无法接受的。考虑下面的例子：

- 在界面的第一屏，打开对话框，用户所看到的数据是被一个特定的 `Session` 和数据 库事务载入(load)的。用户可以随意修改对话框中的数据对象。
- 5分钟后，用户点击“保存”，期望所做出的修改被持久化；同时他也期望自己是唯一修改这个信息的人，不会出现 修改冲突。

从用户的角度来看，我们把这个操作单元称为长时间运行的对话（conversation），或者(or 应用事务,application transaction)。 在你的应用程序中，可以有很多种方法来实现它。

头一个幼稚的做法是，在用户思考的过程中，保持 `Session` 和数据库事务是打开的， 保持数据库锁定，以阻止并发修改，从而保证数据库事务隔离级别和原子操作。这种方式当然是一个反模式， 因为锁争用会导致应用程序无法扩展并发用户的数目。

很明显，我们必须使用多个数据库事务来实现这个对话。在这个例子中，维护业务处理的 事务隔离变成了应用程序层的部分责任。一个对话通常跨越多个数据库事务。如果仅仅只有一 个数据库事务（最后的那个事务）保存更新过的数据，而所有其他事务只是单纯的读取数据（例如在一 个跨越多个请求/响应周期的向导风格的对话框中），那么应用程序事务将保证其原子性。这种方式比听 起来还要容易实现，特别是当你使用了Hibernate的下述特性的时候：

- 自动版本化 - Hibernate能够自动进行乐观并发控制， 如果在用户思考 的过程中发生并发修改，Hibernate能够自动检测到。一般我们只在对话结束时才检查。
- 脱管对象（Detached Objects）- 如果你决定采用前面已经讨论过的 *session-per-request*模式，所有载入的实例在用户思考的过程 中都处于与Session脱离的状态。Hibernate允许你把与Session脱离的对象重新关联到Session 上，并且对修改进行持久化，这种模式被称为 *session-per-request-with-detached-objects*。自动版本化被用来隔离并发修改。
- *Extended (or Long) Session* - Hibernate 的 `Session` 可以在数据库事务提交之后和底层的JDBC连接断开，当一个新的客户端请求到来的时候，它又重新连接上底层的 JDBC连接。这种模式被称之为*session-per-conversation*，这种情况可 能会造成不必要的Session和JDBC连接的重新关联。自动版本化被用来隔离并发修改，`Session` 通常不允许自动flush,而是明确flush。

session-per-request-with-detached-objects 和 *session-per-conversation* 各有优缺点，我们在本章后面乐观并发 控制那部分再进行讨论。

11.1.3. 关注对象标识(Considering object identity)

应用程序可能在两个不同的 `Session` 中并发访问同一持久化状态，但是， 一个持久化类的实例无法在两个 `Session` 中共享。因此有两种不同的标识语义：

数据库标识

`foo.getId().equals( bar.getId() )`

JVM 标识

`foo==bar`

对于那些关联到 特定 `Session` （也就是在单个 `Session` 的范围内）上的对象来说，这 两种标识的语义是等价的，与数据库标识对应的JVM标识是由Hibernate来保 证的。不过，当应用程序在两个不同的session中并发访问具有同一持久化标 识的业务对象实例的时候，这个业务对象的两个实例事实上是不相同的（从 JVM识别来看）。这种冲突可以通过在同步和提交的时候使用自动版本化和乐 观锁定方法来解决。

这种方式把关于并发的头疼问题留给了Hibernate和数据库；由于在单个线程内，操作单元中的对象识别不 需要代价昂贵的锁定或其他意义上的同步，因此它同时可以提供最好的可伸缩性。只要在单个线程只持有一个 `Session` ，应用程序就不需要同步任何业务对象。在 `Session` 的范围内，应用程序可以放心的使用 `==` 进行对象比较。

不过，应用程序在 `Session` 的外面使用 `==` 进行对象比较可能会 导致无法预期的结果。在一些无法预料的场合，例如，如果你把两个脱管对象实例放进同一个 `Set` 的时候，就可能发生。这两个对象实例可能有同一个数据库标识（也就是说， 他们代表了表的同一行数据），从JVM标识的定义上来说，对脱管的对象而言，Hibernate无法保证他们 的的JVM标识一致。开发人员必须覆盖持久化类的 `equals()` 方法和 `hashCode()` 方法，从而实现自定义的对象相等语义。警告：不要使用数据库标识 来实现对象相等，应该使用业务键值，由唯一的，通常不变的属性组成。当一个瞬时对象被持久化的时 候，它的数据库标识会发生改变。如果一个瞬时对象（通常也包括脱管对象实例）被放入一 个 `Set` ，改变它的hashcode会导致与这个 `Set` 的关系中断。虽 然业务键值的属性不象数据库主键那样稳定不变，但是你只需要保证在同一个 `Set` 中的对象属性的稳定性就足够了。请到Hibernate网站去寻求这个问题更多的详细的讨论。请注意，这不是一 个有关Hibernate的问题，而仅仅是一个关于Java对象标识和判等行为如何实现的问题。

11.1.4. 常见问题

决不要使用反模式*session-per-user-session*或者 *session-per-application*（当然，这个规定几乎没有例外）。请注意， 下述一些问题可能也会出现在我们推荐的模式中，在你作出某个设计决定之前，请务必理解该模式的应用前提。

- `Session` 对象是非线程安全的。如果一个 `Session` 实例允许共享的话，那些支持并发运行的东东，例如HTTP request，session beans,或者是 Swing workers，将会导致出现资源争用（race condition）。如果在 `HttpSession` 中有 Hibernate 的 `Session` 的话（稍后讨论），你应该考虑同步访问你的Http session。 否则，只要用户足够快的点击浏览器的“刷新”，就会导致两个并发运行线程使用同一个 `Session` 。

- 一个由Hibernate抛出的异常意味着你必须立即回滚数据库事务，并立即关闭 `Session` （稍后会展开讨论）。如果你的 `Session` 绑定到一个应用程序上，你必 须停止该应用程序。回滚数据库事务并不会把你的业务对象退回到事务启动时候的状态。这 意味着数据库状态和业务对象状态不同步。通常情况下，这不是什么问题，因为异常是不可 恢复的,你必须在回滚之后重新开始执行。

- `Session` 缓存了处于持久化状态的每个对象（Hibernate会监视和检查脏数据）。 这意味着，如果你让 `Session` 打开很长一段时间，或是仅仅载入了过多的数据， `Session` 占用的内存会一直增长，直到抛出OutOfMemoryException异常。这个 问题的一个解决方法是调用 `clear()` 和 `evict()` 来管理 `Session` 的缓存，但是如果你需要大批量数据操作的话，最好考虑 使用存储过程。在第 13 章 批量处理（*Batch processing*）中有一些解决方案。在用户会话期间一直保持 `Session` 打开也意味着出现脏数据的可能性很高。

11.2. 数据库事务声明

数据库（或者系统）事务的声明总是必须的。在数据库事务之外，就无法和数据库通讯（这可能会让那些习惯于 自动提交事务模式的开发人员感到迷惑）。永远使用清晰的事务声明，即使只读操作也是如此。进行 显式的事务声明并不总是需要的，这取决于你的事务隔离级别和数据库的能力，但不管怎么说，声明事务总归有益无害。当然，一个单独的数据库事务总是比很多琐碎的事务性能更好，即时对读数据而言也是一样。

一个Hibernate应用程序可以运行在非托管环境中（也就是独立运行的应用程序，简单Web应用程序， 或者Swing图形桌面应用程序），也可以运行在托管的J2EE环境中。在一个非托管环境中，Hibernate 通常自己负责管理数据库连接池。应用程序开发人员必须手工设置事务声明，换句话说，就是手工启 动，提交，或者回滚数据库事务。一个托管的环境通常提供了容器管理事务(CMT)，例如事务装配通过可声 明的方式定义在EJB session beans的部署描述符中。可编程式事务声明不再需要，即使是 `Session` 的同步也可以自动完成。

让持久层具备可移植性是人们的理想,这种移植发生在非托管的本地资源环境，与依赖JTA但是使用BMT而非CMT的系统之间。在两种情况下你都可以使用编程式的事务管理。Hibernate提供了一套称为 `Transaction` 的封装API， 用来把你的部署环境中的本地事务管理系统转换到Hibernate事务上。这个API是可选的，但是我们强烈 推荐你使用，除非你用CMT session bean。

通常情况下，结束 `Session` 包含了四个不同的阶段:

- 同步session(flush,刷出到磁盘)
- 提交事务
- 关闭session
- 处理异常

session的同步(flush,刷出）前面已经讨论过了，我们现在进一步考察在托管和非托管环境下的事务声明和异常处理。

11.2.1. 非托管环境

如果Hibernat持久层运行在一个非托管环境中，数据库连接通常由Hibernate的简单（即非DataSource)连接池机制 来处理。session/transaction处理方式如下所示：

```
//Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

你不需要显式 `flush()` `Session` - 对 `commit()` 的调用会自动触发session的同步(取决于session的第 10.10 节 “Session刷出(flush)”)。调用 `close()` 标志session的结束。 `close()` 方法重要的暗示是， `session` 释放了JDBC连接。这段Java代码在非托管环境下和JTA环境下都可以运行。

更加灵活的方案是Hibernate内置的"current session"上下文管理，前文已经讲过：

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

你很可能从未在一个通常的应用程序的业务代码中见过这样的代码片断：致命的（系统）异常应该总是 在应用程序“顶层”被捕获。换句话说，执行Hibernate调用的代码（在持久层）和处理 `RuntimeException` 异常的代码（通常只能清理和退出应用程序）应该在不同 的应用程序逻辑层。Hibernate的当前上下文管理可以极大地简化这一设计，你所有的一切就是 `SessionFactory` 。 异常处理将在本章稍后进行讨论。

请注意，你应该选择 `org.hibernate.transaction.JDBCTransactionFactory` (这是默认选项)，对第二个例子来说， `hibernate.current_session_context_class` 应该是 `"thread"`

11.2.2. 使用JTA

如果你的持久层运行在一个应用服务器中（例如，在EJB session beans的后面），Hibernate获取 的每个数据源连接将自动成为全局JTA事务的一部分。 你可以安装一个独立的JTA实现，使用它而不使用EJB。Hibernate提供了两种策略进行JTA集成。

如果你使用bean管理事务（BMT），可以通过使用Hibernate的 `Transaction` API来告诉 应用服务器启动和结束BMT事务。因此，事务管理代码和在非托管环境下是一样的。

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

如果你希望使用与事务绑定的 `Session` ，也就是使用 `getCurrentSession()` 来简化上下文管理，你将不得不直接使用JTA `UserTransaction` API。

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

在CMT方式下，事务声明是在session bean的部署描述符中，而不需要编程。 因此，代码被简化为:

```
// CMT idiom
Session sess = factory.getCurrentSession();

// do some work
...
```

在CMT/EJB中甚至会自动rollback，因为假若有未捕获的 `RuntimeException` 从session bean方法中抛出，这就会通知容器把全局事务回滚。这就意味着，在*BMT*或者*CMT*中，你根本就不需要使用*Hibernate* `Transaction` *API*，你自动得到了绑定到事务的“当前”*Session*。

注意，当你配置Hibernate的transaction factory的时候，在直接使用JTA的时候（BMT），你应该选择 `org.hibernate.transaction.JTATransactionFactory` ,在CMT session bean中选择 `org.hibernate.transaction.CMTTransactionFactory` 。记得也要设置 `hibernate.transaction.manager_lookup_class` 。还有，确认你的 `hibernate.current_session_context_class` 未设置（为了向下兼容），或者设置为 `"jta"` 。

`getCurrentSession()` 在JTA环境中有一个弊端。对 `after_statement` 连接释放方式有一个警告，这是被默认使用的。因为JTA规范的一个很愚蠢的限制，Hibernate不可能自动清理任何未关闭的 `ScrollableResults` 或者 `Iterator` ，它们是由 `scroll()` 或 `iterate()` 产生的。你*must*通过在 `finally` 块中，显式

调用 `ScrollableResults.close()` 或者 `Hibernate.close(Iterator)` 方法来释放底层数据库游标。(当然，大部分程序完全可以很容易的避免在JTA或CMT代码中出现 `scroll()` 或 `iterate()` 。)

11.2.3. 异常处理

如果 `Session` 抛出异常 (包括任何 `SQLException`), 你应该立即回滚数据库事务，调用 `Session.close()` ，丢弃该 `Session` 实例。 `Session` 的某些方法可能会导致session 处于不一致的状态。所有由Hibernate抛出的异常都视为不可以恢复的。确保在 `finally` 代码块中调用 `close()` 方法，以关闭掉 `Session` 。

`HibernateException` 是一个非检查期异常（这不同于Hibernate老的版本）， 它封装了Hibernate持久层可能出现的大多数错误。我们的观点是，不应该强迫应用程序开发人员 在底层捕获无法恢复的异常。在大多数软件系统中，非检查期异常和致命异常都是在相应方法调用 的堆栈的顶层被处理的（也就是说，在软件上面的逻辑层），并且提供一个错误信息给应用软件的用户 （或者采取其他某些相应的操作）。请注意，Hibernate也有可能抛出其他并不属于 `HibernateException` 的非检查期异常。这些异常同样也是无法恢复的，应该 采取某些相应的操作去处理。

在和数据库进行交互时，Hibernate把捕获的 `SQLException` 封装为Hibernate的 `JDBCException` 。事实上，Hibernate尝试把异常转换为更有实际含义的 `JDBCException` 异常的子类。底层的 `SQLException` 可以 通过 `JDBCException.getCause()` 来得到。Hibernate通过使用关联到 `SessionFactory` 上的 `SQLExceptionConverter` 来 把 `SQLException` 转换为一个对应的 `JDBCException` 异常的子类。默认情况下， `SQLExceptionConverter` 可以通过配置dialect 选项指定；此外，也可以使用用户自定义的实现类（参考javadocs `SQLExceptionConverterFactory` 类来了解详情）。标准的 `JDBCException` 子类型是：

- `JDBCConnectionException` - 指明底层的JDBC通讯出现错误
- `SQLGrammarException` - 指明发送的SQL语句的语法或者格式错误
- `ConstraintViolationException` - 指明某种类型的约束违例错误
- `LockAcquisitionException` - 指明了在执行请求操作时，获取 所需的锁级别时出现的错误。
- `GenericJDBCException` - 不属于任何其他种类的原生异常

11.2.4. 事务超时

EJB这样的托管环境有一项极为重要的特性，而它从未在非托管环境中提供过，那就是事务超时。在出现错误的事务行为的时候，超时可以确保不会无限挂起资源、对用户没有交代。在托管(JTA)环境之外，Hibernate无法完全提供这一功能。但是，Hiberante至少可以控制数据访问，确保数据库级别的死锁，和返回巨大结果集的查询被限定在一个规定的时间内。在托管环境中，Hibernate会把事务超时转交给JTA。这一功能通过Hibernate `Transaction` 对象进行抽象。

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

注意 `setTimeout()` 不应该在CMT bean中调用，此时事务超时值应该是被声明式定义的。

11.3. 乐观并发控制(Optimistic concurrency control)

唯一能够同时保持高并发和高可伸缩性的方法就是使用带版本化的乐观并发控制。版本检查使用版本号、 或者时间戳来检测更新冲突（并且防止更新丢失）。Hibernate为使用乐观并发控制的代码提供了三种可 能的方法，应用程序在编写这些代码时，可以采用它们。我们已经在前面应用程序对话那部分展示了 乐观并发控制的应用场景，此外，在单个数据库事务范围内，版本检查也提供了防止更新丢失的好处。

11.3.1. 应用程序级别的版本检查(Application version checking)

未能充分利用Hibernate功能的实现代码中，每次和数据库交互都需要一个新的 Session ，而且开发人员必须在显示数据之前从数据库中重 新载入所有的持久化对象实例。这种方式迫使应用程序自己实现版本检查来确保 对话事务的隔离，从数据访问的角度来说是最低效的。这种使用方式和 entity EJB最相似。

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateExcepti
foo.setProperty("bar");

t.commit();
session.close();
```

version 属性使用 <version> 来映射，如果对象 是脏数据，在同步的时候，Hibernate会自动增加版本号。

当然，如果你的应用是在一个低数据并发环境下，并不需要版本检查的话，你照样可以使用 这种方式，只不过跳过版本检查就是了。在这种情况下，最晚提交生效（*last commit wins*）就是你的长对话的默认处理策略。 请记住这种策略可能会让应用软件的用户感到困惑，因为他们有可能会碰上更新丢失掉却没 有出错信息，或者需要合并更改冲突的情况。

很明显，手工进行版本检查只适合于某些软件规模非常小的应用场景，对于大多数软件应用场景 来说并不现实。通常情况下，不仅是单个对象实例需要进行版本检查，整个被修改过的关 联对象图也都需要进行版本检查。作为标准设计范例，Hibernate使用扩展周期的 Session 的方式，或者脱管对象实例的方式来提供自动版本检查。

11.3.2. 扩展周期的session和自动版本化

单个 `Session` 实例和它所关联的所有持久化对象实例都被用于整个 对话，这被称为*session-per-conversation*。Hibernate在同步的时候进行对象实例的版本检查，如果检测到并发修 改则抛出异常。由开发人员来决定是否需要捕获和处理这个异常（通常的抉择是给用户 提供一个合并更改，或者在无脏数据情况下重新进行业务对话的机会）。

在等待用户交互的时候， `Session` 断开底层的JDBC连接。这种方式 以数据库访问的角度来说是最高效的方式。应用程序不需要关心版本检查或脱管对象实例 的重新关联，在每个数据库事务中，应用程序也不需要载入读取对象实例。

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC co

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

`foo` 对象知道它是在哪个 `Session` 中被装入的。在一个旧session中开启一个新的数据库事务，会导致session获取一个新的连接，并恢复session的功能。将数据库事务提交，使得session从JDBC连接断开，并将此连接交还给连接池。在重新连接之后，要强制对你没有更新的数据进行一次版本检查，你可以对所有可能被其他事务修改过的对象，使用参数 `LockMode.READ` 来调用 `Session.lock()` 。你不用lock任何你正在更新的数据。一般你会在扩展的 `Session` 上设置 `FlushMode.NEVER` ，因此只有最后一个数据库事务循环才会真正的吧整个对话中发生的修改发送到数据库。因此，只有这最后一次数据库事务才会包含 `flush()` 操作，然后在整个对话结束后，还要 `close()` 这个session。

如果在用户思考的过程中， `Session` 因为太大了而不能保存，那么这种模式是有问题的。举例来说，一个 `HttpSession` 应该尽可能的小。由于 `Session` 是一级缓存，并且保持了所有被载入过的对象，因此 我们只应该在那些少量的request/response情况下使用这种策略。你应该只把一个 `Session` 用于单个对话，因为它很快就会出现脏数据。

（注意，早期的Hibernate版本需要明确的对 `Session` 进行disconnec和reconnect。这些方法现在已经过时了，打开事务和关闭事务会起到同样的效果。）

此外，也请注意，你应该让与数据库连接断开的 `Session` 对持久层保持 关闭状态。换句话说，在三层环境中，使用有状态的EJB session bean来持有 `Session` ， 而不要把它传递到web层（甚至把它序列化到一个单独的层），保存在 `HttpSession` 中。

扩展session模式，或者被称为每次对话一个*session(session-per-conversation)*, 在与自动管理当前session上下文联用的时候会更困难。你需要提供你自己的 `CurrentSessionContext` 实现。请参阅Hibernate Wiki以获得示例。

11.3.3. 脱管对象(deatched object)和自动版本化

这种方式下，与持久化存储的每次交互都发生在一个新的 `Session` 中。 然而，同一持久化对象实例可以在多次与数据库的交互中重用。应用程序操纵脱管对象实例的状态，这个脱管对象实例最初是在另一个 `Session` 中载入的，然后 调用 `Session.update()` ，`Session.saveOrUpdate()` , 或者 `Session.merge()` 来重新关联该对象实例。

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been
t.commit();
session.close();
```

Hibernate会再一次在同步的时候检查对象实例的版本，如果发生更新冲突，就抛出异常。

如果你确信对象没有被修改过，你也可以调用 `lock()` 来设置 `LockMode.READ` （绕过所有的缓存，执行版本检查），从而取 代 `update()` 操作。

11.3.4. 定制自动版本化行为

对于特定的属性和集合，通过为它们设置映射属性`optimistic-lock`的值为`false`，来禁止Hibernate的版本自动增加。这样的话，如果该属性 脏数据，Hibernate将不再增加版本号。

遗留系统的数据库Schema通常是静态的，不可修改的。或者，其他应用程序也可能访问同一数据 库，根本无法得知如何处理版本号，甚至时间戳。在以上的所有场景中，实现版本化不能依靠 数据库表的某个特定列。在 <class> 的映射中设置 `optimistic-lock="all"` 可以在没有版本或者时间戳属性映射的情况下实现 版本检查，此时Hibernate将比较一行记录的每个字段的状态。请注意，只有当Hibernate能够比 较新旧状态的情况下，这种方式才能生效，也就是说， 你必须使用单个长生命周期`Session`模式，而不能使用 session-per-request-with-detached-objects模式。

有些情况下，只要更改不发生交错，并发修改也是允许的。当你在 <class> 的映射中设置`optimistic-lock="dirty"`，Hibernate在同步的时候将只比较有脏 数据的字段。

在以上所有场景中，不管是专门设置一个版本/时间戳列，还是进行全部字段/脏数据字段比较， Hibernate都会针对每个实体对象发送一条`UPDATE`（带有相应的`WHERE`语句 ）的SQL语句来执行版本检查和数据更新。如果你对关联实体 设置级联关系使用传播性持久化（transitive persistence），那么Hibernate可能会执行不必 要的update语句。这通常不是个问题，但是数据库里面对*on update*点火 的触发器可能在脱管对象没有任何更改的情况下被触发。因此，你可以在 <class> 的映射中，通过设置`select-before-update="true"` 来定制这一行为，强制Hibernate `SELECT`这个对象实例，从而保证， 在更新记录之前，对象的确是被修改过。

11.4. 悲观锁定(Pessimistic Locking)

用户其实并不需要花很多精力去担心锁定策略的问题。通常情况下，只要为JDBC连接指定一下隔 离级别，然后让数据库去搞定一切就够了。然而，高级用户有时候希望进行一个排它的悲观锁定， 或者在一个新的事务启动的时候，重新进行锁定。

Hibernate总是使用数据库的锁定机制，从不在内存中锁定对象！

类 `LockMode` 定义了Hibernate所需的不同的锁定级别。一个锁定 可以通过以下的机制来设置:

- 当Hibernate更新或者插入一行记录的时候，锁定级别自动设置为 `LockMode.WRITE` 。
- 当用户显式的使用数据库支持的SQL格式 `SELECT ... FOR UPDATE` 发送SQL的时候，锁定级别设置为 `LockMode.UPGRADE`
- 当用户显式的使用Oracle数据库的SQL语句 `SELECT ... FOR UPDATE NOWAIT` 的时候，锁定级别设置 `LockMode.UPGRADE_NOWAIT`
- 当Hibernate在“可重复读”或者是“序列化”数据库隔离级别下读取数据的时候，锁定模式 自动设置为 `LockMode.READ` 。这种模式也可以通过用户显式指定进行设置。
- `LockMode.NONE` 代表无需锁定。在 `Transaction` 结束时， 所有的对象都切换到该模式上来。与session相关联的对象通过调用 `update()` 或者 `saveOrUpdate()` 脱离该模式。

"显式的用户指定"可以通过以下几种方式之一来表示:

- 调用 `Session.load()` 的时候指定 `锁定模式(LockMode)` 。
- 调用 `Session.lock()` 。
- 调用 `Query.setLockMode()` 。

如果在 `UPGRADE` 或者 `UPGRADE_NOWAIT` 锁定模式下调 用 `Session.load()` ，并且要读取的对象尚未被session载入过，那么对象 通过 `SELECT ... FOR UPDATE` 这样的SQL语句被载入。如果为一个对象调用 `load()` 方法时，该对象已经在另一个较少限制的锁定模式下被载入了，那 么Hibernate就对该对象调用 `lock()` 方法。

如果指定的锁定模式是 `READ` ， `UPGRADE` 或 `UPGRADE_NOWAIT` ，那么 `Session.lock()` 就 执行版本号检查。（在 `UPGRADE` 或者 `UPGRADE_NOWAIT` 锁定模式下，执行 `SELECT ... FOR UPDATE` 这样的SQL语句。）

如果数据库不支持用户设置的锁定模式，Hibernate将使用适当的替代模式（而不是扔出异常）。 这一点可以确保应用程序的可移植性。

11.5. 连接释放模式(Connection Release Modes)

Hibernate关于JDBC连接管理的旧(2.x)行为是， `Session` 在第一次需要的时候获取一个连接，在session关闭之前一直会持有这个连接。Hibernate引入了连接释放的概念，来告诉session如何处理它的JDBC连接。注意，下面的讨论只适用于采用配置 `ConnectionProvider` 来提供连接的情况，用户自己提供的连接与这里的讨论无关。通过 `org.hibernate.ConnectionReleaseMode` 的不同枚举值来使用不用的释放模式:

- `ON_CLOSE` - 基本上就是上面提到的老式行为。Hibernate session在第一次需要进行JDBC操作的时候获取连接，然后持有它，直到session关闭。
- `AFTER_TRANSACTION` - 在 `org.hibernate.Transaction` 结束后释放连接。
- `AFTER_STATEMENT` (也被称做积极释放) - 在每一条语句被执行后就释放连接。但假若语句留下了与session相关的资源，那就不会被释放。目前唯一的这种情形就是使用 `org.hibernate.ScrollableResults` 。

`hibernate.connection.release_mode` 配置参数用来指定使用哪一种释放模式。可能的值有：

- `auto` (默认) - 这一选择把释放模式委派给 `org.hibernate.transaction.TransactionFactory.getDefaultRelease` 方法。对JTATransactionFactory来说，它会返回ConnectionReleaseMode.AFTER_STATEMENT;对JDBCTransactionFactory来说，则是ConnectionReleaseMode.AFTER_TRANSACTION。很少需要修改这一默认行为，因为假若设置不当，就会带来bug，或者给用户代码带来误导。
- `on_close` - 使用 ConnectionReleaseMode.ON_CLOSE. 这种方式是为了向下兼容的,但是已经完全不被鼓励使用了。
- `after_transaction` - 使用ConnectionReleaseMode.AFTER_TRANSACTION。这一设置不应该在JTA环境下使用。也要注意，使用ConnectionReleaseMode.AFTER_TRANSACTION的时候，假若session 处于auto-commit状态，连接会像AFTER_STATEMENT那样被释放。
- `after_statement` - 使用ConnectionReleaseMode.AFTER_STATEMENT。除此之外，会查询配置的 `ConnectionProvider` ，是否它支持这一设置（(`supportsAggressiveRelease()`)）。假若不支持，释放模式会被设置为ConnectionReleaseMode.AFTER_TRANSACTION。只有在你每次调用 `ConnectionProvider.getConnection()` 获取底层JDBC连接的时候，都可以确信获得同一个连接的时候，这一设置才是安全的；或者在auto-commit环境中，你可以不管是否每次都获得同一个连接的时候，这才是安全的。

第 12 章 拦截器与事件(Interceptors and events)

目录

应用程序能够响应Hibernate内部产生的特定事件是非常有用的。这样就允许实现某些通用的功能 以及允许对Hibernate功能进行扩展。

12.1. 拦截器(Interceptors)

`Interceptor` 接口提供了从会话(session)回调(callback)应用程序(application)的机制， 这种回调机制可以允许应用程序在持久化对象被保存、更新、删除或是加载之前，检查并（或）修改其 属性。一个可能的用途，就是用来跟踪审核(auditing)信息。例如：下面的这个 `拦截器` ，会在一个实现了 `Auditable` 接口的对象被创建时自动地设置 `createTimestamp` 属性，并在实现了 `Auditable` 接口的对象被更新时，同步更新 `lastUpdateTimestamp` 属性。

你可以直接实现 `Interceptor` 接口，也可以（最好）继承自 `EmptyInterceptor` 。

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i]
                    currentState[i] = new Date();
                    return true;
```

```
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) )
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

拦截器可以有两种: `Session` 范围内的，和 `SessionFactory` 范围内的。

当使用某个重载的SessionFactory.openSession()使用 `Interceptor` 作为参数调用打开一个session的时候，就指定了 `Session` 范围内的拦截器。

```
Session session = sf.openSession( new AuditInterceptor() );
```

`SessionFactory` 范围内的拦截器要通过 `Configuration` 中注册，而这必须在创建 `SessionFactory` 之前。在这种情况下，给出的拦截器会被这个 `SessionFactory` 所打开的所有session使用了；除非session打开时明确指明了使用的拦截器。 `SessionFactory` 范围内的拦截器，必须是线程安全的，因为多个session可能并发使用这个拦截器，要因此小心不要保存与session相关的状态。

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

12.2. 事件系统(Event system)

如果需要响应持久层的某些特殊事件，你也可以使用Hibernate3的事件框架。 该事件系统可以用来替代拦截器，也可以作为拦截器的补充来使用。

基本上， `Session` 接口的每个方法都有相对应的事件。比如 `LoadEvent` ， `FlushEvent` ，等等（查阅XML配置文件 的DTD，以及 `org.hibernate.event` 包来获得所有已定义的事件的列表）。当某个方 法被调用时，Hibernate `Session` 会生成一个相对应的事件并激活所 有配置好的事件监听器。系统预设的监听器实现的处理过程就是被监听的方法要做的（被监听的方法所做的其实仅仅是激活监听器，“实际”的工作是由监听器完成的）。不过，你可以自由地选择实现 一个自己定制的监听器（比如，实现并注册用来处理处理 `LoadEvent` 的 `LoadEventListener` 接口）， 来负责处理所有的调用 `Session` 的 `load()` 方法的请求。

监听器应该被看作是单例(singleton)对象，也就是说，所有同类型的事件的处理共享同一个监听器实例，因此监听器 不应该保存任何状态（也就是不应该使用成员变量）。

用户定制的监听器应该实现与所要处理的事件相对应的接口，或者从一个合适的基类继承（甚至是从Hibernate自带的默认事件监听器类继承， 为了方便你这样做，这些类都被声明成non-final的了）。用户定制的监听器可以通过编程使用 `Configuration` 对象 来注册，也可以在Hibernate的XML格式的配置文件中进行声明（不支持在Properties格式的配置文件声明监听器）。 下面是一个用户定制的加载事件(load event)的监听器：

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener i
    public void onLoad(LoadEvent event, LoadEventListener.LoadType
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(),
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

你还需要修改一处配置，来告诉Hibernate，除了默认的监听器，还要附加选定的监听器。

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEve
        </event>
    </session-factory>
</hibernate-configuration>
```

看看用另一种方式，通过编程的方式来注册它。

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoad
cfg.EventListeners().setLoadEventListeners(stack);
```

通过在XML配置文件声明而注册的监听器不能共享实例。如果在多个 `<listener/>` 节点中使用 了相同的类的名字，则每一个引用都将会产生一个独立的实例。如果你需要在多个监听器类型之间共享 监听器的实例，则你必须使用编程的方式来进行注册。

为什么我们实现了特定监听器的接口，在注册的时候还要明确指出我们要注册哪个事件的监听器呢？ 这是因为一个类可能实现多个监听器的接口。在注册的时候明确指定要监听的事件，可以让启用或者禁用对某个事件的监听的配置工作简单些。

12.3. Hibernate的声明式安全机制

通常，Hibernate应用程序的声明式安全机制由会话外观层(session facade)所管理。 现在，Hibernate3允许某些特定的行为由JACC进行许可管理，由JAAS进行授权管理。 本功能是一个建立在事件框架之上的可选的功能。

首先，你必须要配置适当的事件监听器（event listener），来激活使用JAAS管理授权的功能。

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDele
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpda
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInse
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEv
```

注意， `<listener type="..." class="..."/>` 只是 `<event type="..."><listener class="..."/></event&gt` 的简写，对每一个事件类型都必须严格的有一个监听器与之对应。

接下来，仍然在 `hibernate.cfg.xml` 文件中，绑定角色的权限：

```
<grant role="admin" entity-name="User" actions="insert,update,read"
<grant role="su" entity-name="User" actions="*"/>
```

这些角色的名字就是你的JACC provider所定义的角色的名字。

第 13 章 批量处理（Batch processing）

目录

使用Hibernate将 100 000 条记录插入到数据库的一个很自然的做法可能是这样的

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

这段程序大概运行到 50 000 条记录左右会失败并抛出 `内存溢出异常（OutOfMemoryException）` 。 这是因为 Hibernate 把所有新插入的 `客户（Customer）` 实例在 session级别的缓存区进行了缓存的缘故。

我们会在本章告诉你如何避免此类问题。首先，如果你要执行批量处理并且想要达到一个理想的性能， 那么使用JDBC的批量（batching）功能是至关重要。将JDBC的批量抓取数量（batch size）参数设置到一个合适值 （比如，10-50之间）：

```
hibernate.jdbc.batch_size 20
```

注意,假若你使用了 `identiy` 标识符生成器,Hibernate在JDBC级别透明的关闭插入语句的批量执行。

你也可能想在执行批量处理时关闭二级缓存：

```
hibernate.cache.use_second_level_cache false
```

但是，这不是绝对必须的，因为我们可以显式设置 `CacheMode` 来关闭与二级缓存的交互。

13.1. 批量插入（Batch inserts）

如果要将很多对象持久化，你必须通过经常的调用 `flush()` 以及稍后调用 `clear()` 来控制第一级缓存的大小。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size //20,与J
        //flush a batch of inserts and release memory:
        //将本批插入的对象立即写入数据库并释放内存
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.2. 批量更新（Batch updates）

此方法同样适用于检索和更新数据。此外，在进行会返回很多行数据的查询时， 你需要使用 `scroll()` 方法以便充分利用服务器端游标所带来的好处。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.3. StatelessSession (无状态session)接口

作为选择，Hibernate提供了基于命令的API，可以用detached object的形式把数据以流的方法加入到数据库，或从数据库输出。`StatelessSession` 没有持久化上下文，也不提供多少高层的生命周期语义。特别是，无状态session不实现第一级cache,也不和第二级缓存，或者查询缓存交互。它不实现事务化写，也不实现脏数据检查。用stateless session进行的操作甚至不级联到关联实例。stateless session忽略集合类(Collections)。通过stateless session进行的操作不触发Hibernate的事件模型和拦截器。无状态session对数据的混淆现象免疫，因为它没有第一级缓存。无状态session是低层的抽象，和低层JDBC相当接近。

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

注意在上面的例子中，查询返回的 `Customer` 实例立即被脱管(detach)。它们与任何持久化上下文都没有关系。

`StatelessSession` 接口定义的 `insert()`, `update()` 和 `delete()` 操作是直接的数据库行级别操作，其结果是立刻执行一条 `INSERT`, `UPDATE` 或 `DELETE` 语句。因此，它们的语义和 `Session` 接口定义的 `save()`, `saveOrUpdate()` 和 `delete()` 操作有很大的不同。

13.4. DML(数据操作语言)风格的操作(DML-style operations)

hence manipulating (using the SQL `Data Manipulation Language` (DML) statements: `INSERT`, `UPDATE`, `DELETE`) data directly in the database will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style DML statement execution which are performed through the Hibernate Query Language (第 14 章 *HQL: Hibernate查询语言*). 就像已经讨论的那样，自动和透明的 对象/关系 映射（object/relational mapping）关注于管理对象的状态。 这就意味着对象的状态存在于内存，因此直接操作 (使用 SQL `Data Manipulation Language` (DML,数据操作语言）语句 ：`INSERT`, `UPDATE` 和 `DELETE`) 数据库中的数据将不会影响内存中的对象状态和对象数据。 不过，Hibernate提供通过Hibernate查询语言（第 14 章 *HQL: Hibernate查询语言*）来执行大批 量SQL风格的DML语句的方法。

`UPDATE` 和 `DELETE` 语句的语法为：`( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?` 有几点说明：

- 在FROM子句（from-clause）中，FROM关键字是可选的
- 在FROM子句（from-clause）中只能有一个实体名，它可以是别名。如果实体名是别名，那么任何被引用的属性都必须加上此别名的前缀；如果不是别名，那么任何有前缀的属性引用都是非法的。
- 不能在大批量HQL语句中使用第 14.4 节 “join 语法的形式”（显式或者隐式的都不行）。不过在WHERE子句中可以使用子查询。可以在where子句中使用子查询，子查询本身可以包含join。
- 整个WHERE子句是可选的。

举个例子，使用 `Query.executeUpdate()` 方法执行一个HQL `UPDATE` 语句(：(方法命名是来源于JDBC's `PreparedStatement.executeUpdate()`):

```
Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();

        String hqlUpdate = "update Customer c set c.name = :newName
        // or String hqlUpdate = "update Customer set name = :newNa
        int updatedEntities = s.createQuery( hqlUpdate )
                .setString( "newName", newName )
                .setString( "oldName", oldName )
                .executeUpdate();
        tx.commit();
        session.close();
```

HQL `UPDATE` 语句，默认不会影响更新实体的第 5.1.7 节 “版本（version）(可选)”或者第 5.1.8 节 “timestamp (可选)”属性值。这和EJB3规范是一致的。但是，通过使用 `versioned update`，你可以强制Hibernate正确的重置 `version` 或者 `timestamp` 属性值。这通过在 `UPDATE` 关键字后面增加 `VERSIONED` 关键字来实现的。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = 
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

注意，自定义的版本类型(`org.hibernate.usertype.UserVersionType`)不允许和 `update versioned` 语句联用。

执行一个HQL `DELETE`，同样使用 `Query.executeUpdate()` 方法:

```
Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();

        String hqlDelete = "delete Customer c where c.name = :oldNa
        // or String hqlDelete = "delete Customer where name = :old
        int deletedEntities = s.createQuery( hqlDelete )
                .setString( "oldName", oldName )
                .executeUpdate();
        tx.commit();
        session.close();
```

由 `Query.executeUpdate()` 方法返回的 `整型` 值表明了受此操作影响的记录数量。 注意这个数值可能与数据库中被（最后一条SQL语句）影响了的“行”数有关，也可能没有。一个大批量HQL操作可能导致多条实际的SQL语句被执行， 举个例子，对joined-subclass映射方式的类进行的此类操作。这个返回值代表了实际被语句影响了的记录数量。在那个joined-subclass的例子中， 对一个子类的删除实际上可能不仅仅会删除子类映射到的表而且会影响“根”表，还有可能影响与之有继承关系的joined-subclass映射方式的子类的表。

`INSERT` 语句的伪码是: `INSERT INTO EntityName properties_list select_statement`. 要注意的是:

- 只支持INSERT INTO ... SELECT ...形式,不支持INSERT INTO ... VALUES ...形式.

properties_list和SQL `INSERT` 语句中的 `字段定义(column speficiation)` 类似。对参与继承树映射的实体而言，只有直接定义在给定的类级别的属性才能直接在properties_list中使用。超类的属性不被支持；子类的属性无意义。换句话说， `INSERT` 天生不支持多态。

- select*statement*可以是任何合法的*HQL*选择查询，不过要保证返回类型必须和要插入的类型完全匹配。目前，这一检查是在查询编译的时候进行的，而不是把它交给数据库。注意，在*Hibernate* `Type` 间如果只是等价（equivalent）而非相等(equal)_，会导致问题。定义为 `org.hibernate.type.DateType` 和 `org.hibernate.type.TimestampTyp` 的两个属性可能会产生类型不匹配错误，虽然数据库级可能不加区分或者可以处理这种转换。
- 对id属性来说,insert语句给你两个选择。你可以明确地在properties_list表中指定id属性（这样它的值是从对应的select表达式中获得），或者在properties_list中省略它（此时使用生成指）。后一种选择只有当使用在数据库中生成值的id产生器时才能使用；如果是“内存”中计算的类型生成器，在解析时会抛出一个异常。注意，为了说明这一问题，数据库产生值的生成器是 `org.hibernate.id.SequenceGenerator` （和它的子类），以及任何 `org.hibernate.id.PostInsertIdentifierGenerator` 接口的实现。这儿最值得注意的意外是 `org.hibernate.id.TableHiLoGenerator` ，它不能在此使用，因为它没有得到其值的途径。
- 对映射为 `version` 或 `timestamp` 的属性来说，insert语句也给你两个选择，你可以在properties_list表中指定（此时其值从对应的select表达式中获得），或者在properties_list中省略它（此时，使用在 `org.hibernate.type.VersionType` 中定义的 `seed value(种子值)` ）。

执行HQL `INSERT` 语句的例子如下：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

第 14 章 HQL: Hibernate查询语言

目录

Hibernate配备了一种非常强大的查询语言，这种语言看上去很像SQL。但是不要被语法结构 上的相似所迷惑，HQL是非常有意识的被设计为完全面向对象的查询，它可以理解如继承、多态 和关联之类的概念。

14.1. 大小写敏感性问题

除了Java类与属性的名称外，查询语句对大小写并不敏感。 所以 `SeLeCT` 与 `sELEct` 以及 `SELECT` 是相同的，但是 `org.hibernate.eg.FOO` 并不等价于 `org.hibernate.eg.Foo` 并且 `foo.barSet` 也不等价于 `foo.BARSET` 。

本手册中的HQL关键字将使用小写字母. 很多用户发现使用完全大写的关键字会使查询语句 的可读性更强, 但我们发现，当把查询语句嵌入到Java语句中的时候使用大写关键字比较难看。

14.2. from子句

Hibernate中最简单的查询语句的形式如下：

```
from eg.Cat
```

该子句简单的返回 `eg.Cat` 类的所有实例。 通常我们不需要使用类的全限定名, 因为 `auto-import` （自动引入） 是缺省的情况。 所以我们几乎只使用如下的简单写法：

```
from Cat
```

大多数情况下, 你需要指定一个别名, 原因是你可能需要 在查询语句的其它部分引用到 `Cat`

```
from Cat as cat
```

这个语句把别名 `cat` 指定给类 `Cat` 的实例, 这样我们就可以在随后的查询中使用此别名了。 关键字 `as` 是可选的，我们也可以这样写:

```
from Cat cat
```

子句中可以同时出现多个类, 其查询结果是产生一个笛卡儿积或产生跨表的连接。

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

查询语句中别名的开头部分小写被认为是实践中的好习惯， 这样做与Java变量的命名标准保持了一致 (比如， `domesticCat`)。

14.3. 关联(Association)与连接(Join)

我们也可以为相关联的实体甚至是对一个集合中的全部元素指定一个别名, 这时要使用关键字 `join` 。

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

受支持的连接类型是从ANSI SQL中借鉴来的。

- `inner join` （内连接）
- `left outer join` （左外连接）
- `right outer join` （右外连接）
- `full join` (全连接，并不常用)

语句 `inner join` , `left outer join` 以及 `right outer join` 可以简写。

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

通过HQL的 `with` 关键字，你可以提供额外的join条件。

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

还有，一个"fetch"连接允许仅仅使用一个选择语句就将相关联的对象或一组值的集合随着他们的父对象的初始化而被初始化，这种方法在使用到集合的情况下尤其有用，对于关联和集合来说，它有效的代替了映射文件中的外联接 与延迟声明（lazy declarations）. 查看 第 19.1 节 “ 抓取策略(Fetching strategies) ” 以获得等多的信息。

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

一个fetch连接通常不需要被指定别名, 因为相关联的对象不应当被用在 `where` 子句 (或其它任何子句)中。同时，相关联的对象 并不在查询的结果中直接返回，但可以通过他们的父对象来访问到他们。

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

假若使用 `iterate()` 来调用查询，请注意 `fetch` 构造是不能使用的(`scroll()` 可以使用)。 `fetch` 也不应该与 `setMaxResults()` 或 `setFirstResult()` 共用，这是因为这些操作是基于结果集的，而在预先抓取集合类时可能包含重复的数据，也就是说无法预先知道精确的行数。 `fetch` 还不能与独立的 `with` 条件一起使用。通过在一次查询中fetch多个集合，可以制造出笛卡尔积，因此请多加注意。对bag映射来说，同时join fetch多个集合角色可能在某些情况下给出并非预期的结果，也请小心。最后注意，使用 `full join fetch` 与 `right join fetch` 是没有意义的。

如果你使用属性级别的延迟获取（lazy fetching）（这是通过重新编写字节码实现的），可以使用 `fetch all properties` 来强制Hibernate立即取得那些原本需要延迟加载的属性（在第一个查询中）。

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like
```

14.4. join 语法的形式

HQL支持两种关联join的形式： `implicit(隐式)` 与 `explicit（显式）` 。

上一节中给出的查询都是使用 `explicit(显式)` 形式的，其中form子句中明确给出了join关键字。这是建议使用的方式。

`implicit（隐式）` 形式不使用join关键字。关联使用"点号"来进行“引用”。 `implicit` join可以在任何HQL子句中出现. `implicit` join在最终的SQL语句中以inner join的方式出现。

```
from Cat as cat where cat.mate.name like '%s%'
```

14.5. select子句

`select` 子句选择将哪些对象与属性返 回到查询结果集中. 考虑如下情况:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

该语句将选择 `mate` s of other `Cat` s。（其他猫的配偶） 实际上, 你可以更简洁的用以下的查询语句表达相同的含义:

```
select cat.mate from Cat cat
```

查询语句可以返回值为任何类型的属性，包括返回类型为某种组件(Component)的属性:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

查询语句可以返回多个对象和（或）属性，存放在 `Object[]` 队列中,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

或存放在一个 `List` 对象中,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

也可能直接返回一个实际的类型安全的Java对象,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

假设类 `Family` 有一个合适的构造函数.

你可以使用关键字 `as` 给“被选择了的表达式”指派别名:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as
from Cat cat
```

这种做法在与子句 `select new map` 一起使用时最有用:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, cou
from Cat cat
```

该查询返回了一个 `Map` 的对象，内容是别名与被选择的值组成的名-值映射。

14.6. 聚集函数

HQL查询甚至可以返回作用于属性之上的聚集函数的计算结果:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat
from Cat cat
```

受支持的聚集函数如下：

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

你可以在选择子句中使用数学操作符、连接以及经过验证的SQL函数：

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

关键字 `distinct` 与 `all` 也可以使用，它们具有与SQL相同的语义.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

14.7. 多态查询

一个如下的查询语句:

```
from Cat as cat
```

不仅返回 `Cat` 类的实例, 也同时返回子类 `DomesticCat` 的实例. Hibernate 可以在 `from` 子句中指定任何 Java 类或接口. 查询会返回继承了该类的所有持久化子类 的实例或返回声明了该接口的所有持久化类的实例。下面的查询语句返回所有的被持久化的对象：

```
from java.lang.Object o
```

接口 `Named` 可能被各种各样的持久化类声明：

```
from Named n, Named m where n.name = m.name
```

注意，最后的两个查询将需要超过一个的SQL `SELECT`.这表明 `order by` 子句 没有对整个结果集进行正确的排序. (这也说明你不能对这样的查询使用 `Query.scroll()` 方法.)

14.8. where子句

`where` 子句允许你将返回的实例列表的范围缩小. 如果没有指定别名，你可以使用属性名来直接引用属性:

```
from Cat where name='Fritz'
```

如果指派了别名，需要使用完整的属性名:

```
from Cat as cat where cat.name='Fritz'
```

返回名为（属性name等于）'Fritz'的 `Cat` 类的实例。

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

将返回所有满足下面条件的 `Foo` 类的实例： 存在如下的 `bar` 的一个实例，其 `date` 属性等于 `Foo` 的 `startDate` 属性。 复合路径表达式使得 `where` 子句非常的强大，考虑如下情况：

```
from Cat cat where cat.mate.name is not null
```

该查询将被翻译成为一个含有表连接（内连接）的SQL查询。如果你打算写像这样的查询语句

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

在SQL中，你为达此目的将需要进行一个四表连接的查询。

`=` 运算符不仅可以被用来比较属性的值，也可以用来比较实例：

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

特殊属性（小写） `id` 可以用来表示一个对象的唯一的标识符。（你也可以使用该对象的属性名。）

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

第二个查询是有效的。此时不需要进行表连接！

同样也可以使用复合标识符。比如 `Person` 类有一个复合标识符，它由 `country` 属性 与 `medicareNumber` 属性组成。

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

第二个查询也不需要进行表连接。

同样的，特殊属性 `class` 在进行多态持久化的情况下被用来存取一个实例的鉴别值（discriminator value）。 一个嵌入到where子句中的Java类的名字将被转换为该类的鉴别值。

```
from Cat cat where cat.class = DomesticCat
```

你也可以声明一个属性的类型是组件或者复合用户类型（以及由组件构成的组件等等）。永远不要尝试使用以组件类型来结尾的路径表达式（path-expression）（与此相反，你应当使用组件的一个属性来结尾）。 举例来说，如果 `store.owner` 含有一个包含了组件的实体 `address`

```
store.owner.address.city    // 正确
store.owner.address         // 错误!
```

一个“任意”类型有两个特殊的属性 `id` 和 `class`，来允许我们按照下面的方式表达一个连接（ `AuditLog.item` 是一个属性，该属性被映射为 `<any>` ）。

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

注意，在上面的查询与句中， `log.item.class` 和 `payment.class` 将涉及到完全不同的数据库中的列。

14.9. 表达式

在 `where` 子句中允许使用的表达式包括 大多数你可以在SQL使用的表达式种类:

- 数学运算符 `+, -, *, /`
- 二进制比较运算符 =, >=, <=, <>, !=, like
- 逻辑运算符 `and, or, not`
- `in` , `not in` , `between` , `is null` , `is not null` , `is empty` , `is not empty` , `member of` and `not member of`
- "简单的" case, `case ... when ... then ... else ... end` ,和 "搜索" case, `case when ... then ... else ... end`
- 字符串连接符 `...||...` or `concat(...,...)`
- `current_date()` , `current_time()` , `current_timestamp()`
- `second(...)` , `minute(...)` , `hour(...)` , `day(...)` , `month(...)` , `year(...)` ,
- EJB-QL 3.0定义的任何函数或操作： `substring(), trim(), lower(), upper(), length(), locate(), a`
- `coalesce()` 和 `nullif()`
- `str()` 把数字或者时间值转换为可读的字符串
- `cast(... as ...)` , 其第二个参数是某Hibernate类型的名字，以及 `extract(... from ...)` ，只要ANSI `cast()` 和 `extract()` 被底层数据库支持
- HQL `index()` 函数，作用于join的有序集合的别名。
- HQL函数，把集合作为参数: `size(), minelement(), maxelement(), minindex(), maxindex()` ,还有特别的 `elements()` 和 `indices` 函数，可以与数量词加以限定： `some, all, exists, any, in` 。
- 任何数据库支持的SQL标量函数，比如 `sign()` , `trunc()` , `rtrim()` , `sin()`
- JDBC风格的参数传入 `?`
- 命名参数 `:name` , `:start_date` , `:x1`
- SQL 直接常量 `'foo'` , `69` , `6.66E+2` , `'1970-01-01 10:00:01.0'`

- Java `public static final` 类型的常量 `eg.Color.TABBY`

关键字 `in` 与 `between` 可按如下方法使用:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

而且否定的格式也可以如下书写：

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

同样, 子句 `is null` 与 `is not null` 可以被用来测试空值(null).

在Hibernate配置文件中声明HQL“查询替代（query substitutions）”之后， 布尔表达式（Booleans）可以在其他表达式中轻松的使用:

```
<property name="hibernate.query.substitutions">true 1, false 0</pro
```

系统将该HQL转换为SQL语句时，该设置表明将用字符 `1` 和 `0` 来 取代关键字 `true` 和 `false`:

```
from Cat cat where cat.alive = true
```

你可以用特殊属性 `size`, 或是特殊函数 `size()` 测试一个集合的大小。

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

对于索引了（有序）的集合，你可以使用 `minindex` 与 `maxindex` 函数来引用到最小与最大的索引序数。 同理，你可以使用 `minelement` 与 `maxelement` 函数来 引用到一个基本数据类型的集合中最小与最大的元素。

```
from Calendar cal where maxelement(cal.holidays) > current_date
```

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

在传递一个集合的索引集或者是元素集(`elements` 与 `indices` 函数) 或者传递一个子查询的结果的时候，可以使用SQL函数 `any, some, all, exists, in`

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3 > all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

注意，在Hibernate3种，这些结构变量- `size` , `elements` , `indices` , `minindex` , `maxindex` , `minelement` , `maxelement` - 只能在where子句中使用。

一个被索引过的（有序的）集合的元素(arrays, lists, maps)可以在其他索引中被引用（只能在where子句中）：

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

在 `[]` 中的表达式甚至可以是一个算数表达式。

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

对于一个一对多的关联（one-to-many association）或是值的集合中的元素， HQL也提供内建的 `index()` 函数，

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

如果底层数据库支持标量的SQL函数，它们也可以被使用

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

如果你还不能对所有的这些深信不疑，想想下面的查询。如果使用SQL，语句长度会增长多少，可读性会下降多少：

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

提示*:* 会像如下的语句

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_(
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

14.10. order by子句

查询返回的列表(list)可以按照一个返回的类或组件（components)中的任何属性（property）进行排序：

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

可选的 `asc` 或 `desc` 关键字指明了按照升序或降序进行排序.

14.11. group by子句

一个返回聚集值(aggregate values)的查询可以按照一个返回的类或组件（components)中的任何属性（property）进行分组：

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

`having` 子句在这里也允许使用.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

如果底层的数据库支持的话(例如不能在MySQL中使用)，SQL的一般函数与聚集函数也可以出现 在 `having` 与 `order by` 子句中。

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

注意 `group by` 子句与 `order by` 子句中都不能包含算术表达式（arithmetic expressions）. 也要注意Hibernate目前不会扩展group的实体,因此你不能写 `group by cat` ,除非 `cat` 的所有属性都不是聚集的(non-aggregated)。你必须明确的列出所有的非聚集属性。

14.12. 子查询

对于支持子查询的数据库，Hibernate支持在查询中使用子查询。一个子查询必须被圆括号包围起来（经常是SQL聚集函数的圆括号）。 甚至相互关联的子查询（引用到外部查询中的别名的子查询）也是允许的。

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

注意，HQL自查询只可以在select或者where子句中出现。

在select列表中包含一个表达式以上的子查询，你可以使用一个元组构造符（tuple constructors）：

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

注意在某些数据库中（不包括Oracle与HSQL），你也可以在其他语境中使用元组构造符，比如查询用户类型的组件与组合：

```
from Person where name = ('Gavin', 'A', 'King')
```

该查询等价于更复杂的：

```
from Person where name.first = 'Gavin' and name.initial = 'A' and n
```

有两个很好的理由使你不应当作这样的事情：首先，它不完全适用于各个数据库平台；其次，查询现在依赖于映射文件中属性的顺序。

14.13. HQL示例

Hibernate查询可以非常的强大与复杂。实际上，Hibernate的一个主要卖点就是查询语句的威力。这里有一些例子，它们与我在最近的 一个项目中使用的查询非常相似。注意你能用到的大多数查询比这些要简单的多！

下面的查询对于某个特定的客户的所有未支付的账单，在给定给最小总价值的情况下，返回订单的id，条目的数量和总价值， 返回值按照总价值的结果进行排序。为了决定价格，查询使用了当前目录。作为转换结果的SQL查询，使用了 `ORDER` ， `ORDER_LINE` ， `PRODUCT` ， `CATALOG` 和 `PRICE` 库表。

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

这简直是一个怪物！实际上，在现实生活中，我并不热衷于子查询，所以我的查询语句看起来更像这个：

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

下面一个查询计算每一种状态下的支付的数目，除去所有处于 `AWAITING_APPROVAL` 状态的支付，因为在该状态下 当前的用户作出了状态的最新改变。该查询被转换成含有两个内连接以及一个相关联的子选择的SQL查询，该查询使用了表 `PAYMENT` , `PAYMENT_STATUS` 以及 `PAYMENT_STATUS_CHANGE` 。

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

如果我把 `statusChanges` 实例集映射为一个列表（list）而不是一个集合（set）, 书写查询语句将更加简单.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].use
group by status.name, status.sortOrder
order by status.sortOrder
```

下面一个查询使用了MS SQL Server的 `isNull()` 函数用以返回当前用户所属组织的组织帐号及组织未支付的账。 它被转换成一个对表 `ACCOUNT` , `PAYMENT` , `PAYMENT_STATUS` , `ACCOUNT_TYPE` , `ORGANIZATION` 以及 `ORG_USER` 进行的三个内连接， 一个外连接和一个子选择的SQL查询。

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, P
order by account.type.sortOrder, account.accountNumber, payment.due
```

对于一些数据库，我们需要弃用（相关的）子选择。

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, P
order by account.type.sortOrder, account.accountNumber, payment.due
```

14.14. 批量的UPDATE和DELETE

HQL现在支持 `update`，`delete` 和 `insert ... select ...`语句. 查阅 第 13.4 节 “DML(数据操作语言)风格的操作(DML-style operations)” 以获得更多信息。

14.15. 小技巧 & 小窍门

你可以统计查询结果的数目而不必实际的返回他们：

```
( (Integer) session.iterate("select count(*) from ....").next() ).i
```

若想根据一个集合的大小来进行排序，可以使用如下的语句：

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

如果你的数据库支持子选择，你可以在你的查询的where子句中为选择的大小（selection size）指定一个条件:

```
from User usr where size(usr.messages) >= 1
```

如果你的数据库不支持子选择语句，使用下面的查询：

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

因为内连接（inner join）的原因，这个解决方案不能返回含有零个信息的 `User` 类的实例, 所以这种情况下使用下面的格式将是有帮助的:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

JavaBean的属性可以被绑定到一个命名查询（named query）的参数上：

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name a
q.setProperties(fooBean); // fooBean包含方法getName()与getSize()
List foos = q.list();
```

通过将接口 `Query` 与一个过滤器（filter）一起使用，集合（Collections）是可以分页的：

```
Query q = s.createFilter( collection, "" ); // 一个简单的过滤器
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

通过使用查询过滤器（query filter）可以将集合（Collection）的原素分组或排序:

```
Collection orderedCollection = s.filter( collection, "order by this
Collection counts = s.filter( collection, "select this.type, count(
```

不用通过初始化，你就可以知道一个集合（Collection）的大小：

```
( (Integer) session.iterate("select count(*) from ....").next() ).i
```

第 15 章 条件查询(Criteria Queries)

目录

具有一个直观的、可扩展的条件查询API是Hibernate的特色。

15.1. 创建一个 `Criteria` 实例

`org.hibernate.Criteria`接口表示特定持久类的一个查询。`Session` 是 `Criteria` 实例的工厂。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

15.2. 限制结果集内容

一个单独的查询条件是 `org.hibernate.criterion.Criterion` 接口的一个实例。 `org.hibernate.criterion.Restrictions` 类 定义了获得某些内置 `Criterion` 类型的工厂方法。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

约束可以按逻辑分组。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", '
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Hibernate提供了相当多的内置criterion类型(`Restrictions` 子类), 但是尤其有用的是可以允许你直接使用SQL。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like low
    .list();
```

`{alias}` 占位符应当被替换为被查询实体的列别名。

`Property` 实例是获得一个条件的另外一种途径。你可以通过调用 `Property.forName()` 创建一个 `Property` 。

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi
    .list();
```

15.3. 结果集排序

你可以使用 `org.hibernate.criterion.Order` 来为查询结果排序。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

15.4. 关联

你可以使用 `createCriteria()` 非常容易的在互相关联的实体间建立 约束。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

注意第二个 `createCriteria()` 返回一个新的 `Criteria` 实例，该实例引用 `kittens` 集合中的元素。

接下来，替换形态在某些情况下也是很有用的。

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` 并不创建一个新的 `Criteria` 实例。)

`Cat` 实例所保存的之前两次查询所返回的kittens集合是 没有被条件预过滤的。如果你希望只获得符合条件的kittens， 你必须使用 `ResultTransformer` 。

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

15.5. 动态关联抓取

你可以使用 `setFetchMode()` 在运行时定义动态关联抓取的语义。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

这个查询可以通过外连接抓取 `mate` 和 `kittens` 。 查看第 19.1 节 “ 抓取策略 (Fetching strategies) ”可以获得更多信息。

15.6. 查询示例

`org.hibernate.criterion.Example` 类允许你通过一个给定实例 构建一个条件查询。

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

版本属性、标识符和关联被忽略。默认情况下值为null的属性将被排除。

你可以自行调整 `Example` 使之更实用。

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string co
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

你甚至可以使用examples在关联对象上放置条件。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

15.7. 投影(Projections)、聚合（aggregation）和分组（grouping）

`org.hibernate.criterion.Projections` 是 `Projection` 的实例工厂。我们通过调用 `setProjection()` 应用投影到一个查询。

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

在一个条件查询中没有必要显式的使用 "group by" 。某些投影类型就是被定义为分组投影，他们也出现在SQL的 `group by` 子句中。

你可以选择把一个别名指派给一个投影，这样可以使投影值被约束或排序所引用。下面是两种不同的实现方式：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("c
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

`alias()` 和 `as()` 方法简便的将一个投影实例包装到另外一个 别名的 `Projection` 实例中。简而言之，当你添加一个投影到一个投影列表中时 你可以为它指定一个别名：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

你也可以使用 `Property.forName()` 来表示投影：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

15.8. 离线(detached)查询和子查询

`DetachedCriteria` 类使你在一个session范围之外创建一个查询，并且可以使用任意的 `Session` 来执行它。

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(1
txn.commit();
session.close();
```

`DetachedCriteria` 也可以用以表示子查询。条件实例包含子查询可以通过 `Subqueries` 或者 `Property` 获得。

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight).gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

甚至相互关联的子查询也是有可能的：

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.cl
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight).gt(avgWeightForSex) )
    .list();
```

15.9. 根据自然标识查询(Queries by natural identifier)

对大多数查询，包括条件查询而言，因为查询缓存的失效(invalidation)发生得太频繁，查询缓存不是非常高效。然而，有一种特别的查询，可以通过不变的自然键优化缓存的失效算法。在某些应用中，这种类型的查询比较常见。条件查询API对这种用例提供了特别规约。

首先，你应该对你的entity使用 `<natural-id>` 来映射自然键，然后打开第二级缓存。

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

注意,此功能对具有*mutable*自然键的entity并不适用。

然后，打开Hibernate 查询缓存。

现在，我们可以用 `Restrictions.naturalId()` 来使用更加高效的缓存算法。

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

第 16 章 Native SQL查询

目录

你也可以使用你的数据库的Native SQL语言来查询数据。这对你在要使用数据库的某些特性的时候(比如说在查询提示或者Oracle中的 `CONNECT` 关键字)，这是非常有用的。这就能够扫清你把原来直接使用SQL/JDBC 的程序迁移到基于 Hibernate 应用的道路上的障碍。

Hibernate3允许你使用手写的sql来完成所有的create,update,delete,和load操作（包括存储过程）

16.1. 使用 `SQLQuery`

对原生SQL查询执行的控制是通过 `SQLQuery` 接口进行的，通过执行 `Session.createSQLQuery()` 获取这个接口。下面来描述如何使用这个API进行查询。

16.1.1. 标量查询（Scalar queries）

最基本的SQL查询就是获得一个标量（数值）的列表。

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list()
```

它们都将返回一个Object数组(Object[])组成的List，数组每个元素都是CATS表的一个字段值。Hibernate会使用ResultSetMetadata来判定返回的标量值的实际顺序和类型。

如果要避免过多的使用 `ResultSetMetadata` ,或者只是为了更加明确的指名返回值，可以使用 `addScalar()` 。

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

这个查询指定了:

- SQL查询字符串
- 要返回的字段和类型

它仍然会返回Object数组,但是此时不再使用 `ResultSetMetdata` ,而是明确的将ID,NAME和BIRTHDATE按照Long,String和Short类型从resultset中取出。同时，也指明了就算query是使用 `*` 来查询的，可能获得超过列出的这三个字段，也仅仅会返回这三个字段。

对全部或者部分的标量值不设置类型信息也是可以的。

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

基本上这和前面一个查询相同,只是此时使用 `ResultSetMetaData` 来决定NAME和BIRTHDATE的类型，而ID的类型是明确指出的。

关于从ResultSetMetaData返回的java.sql.Types是如何映射到Hibernate类型，是由方言(Dialect)控制的。假若某个指定的类型没有被映射，或者不是你所预期的类型，你可以通过Dialet的 `registerHibernateType` 调用自行定义。

16.1.2. 实体查询(Entity queries)

上面的查询都是返回标量值的，也就是从resultset中返回的“裸”数据。下面展示如何通过 `addEntity()` 让原生查询返回实体对象。

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEnti
```

这个查询指定：

- SQL查询字符串
- 要返回的实体

假设Cat被映射为拥有ID,NAME和BIRTHDATE三个字段的类，以上的两个查询都返回一个List，每个元素都是一个Cat实体。

假若实体在映射时有一个 `many-to-one` 的关联指向另外一个实体，在查询时必须也返回那个实体，否则会导致发生一个"column not found"的数据库错误。这些附加的字段可以使用*标注来自动返回，但我们希望还是明确指明，看下面这个具有指向 `Dog` 的 `many-to-one` 的例子：

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS")
```

这样cat.getDog()就能正常运作。

16.1.3. 处理关联和集合类(Handling associations and collections)

通过提前抓取将 `Dog` 连接获得，而避免初始化proxy带来的额外开销也是可能的。这是通过 `addJoin()` 方法进行的，这个方法可以让你将关联或集合连接进来。

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

上面这个例子中，返回的 `Cat` 对象，其 `dog` 属性被完全初始化了，不再需要数据库的额外操作。注意，我们加了一个别名("cat")，以便指明join的目标属性路径。通过同样的提前连接也可以作用于集合类，例如，假若 `Cat` 有一个指向 `Dog` 的一对多关联。

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

到此为止，我们碰到了天花板：若不对SQL查询进行增强，这些已经是在Hibernate中使用原生SQL查询所能做到的最大可能了。下面的问题即将出现：返回多个同样类型的实体怎么办？或者默认的别名/字段不够又怎么办？

16.1.4. 返回多个实体(Returning multiple entities)

到目前为止,结果集字段名被假定为和映射文件中指定的的字段名是一致的。假若SQL查询连接了多个表，同一个字段名可能在多个表中出现多次，这就会造成问题。

下面的查询中需要使用字段别名注射（这个例子本身会失败）：

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.M
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询的本意是希望每行返回两个Cat实例，一个是cat,另一个是它的妈妈。但是因为它们的字段名被映射为相同的，而且在某些数据库中，返回的字段别名是“c.ID”,"c.NAME"这样的形式，而它们和在映射文件中的名字（"ID"和"NAME"）不匹配，这就会造成失败。

下面的形式可以解决字段名重复：

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询指明：

- SQL查询语句，其中包含占位附来让Hibernate注射字段别名
- 查询返回的实体

上面使用的{cat.*}和*{mother.*}*标记是作为“所有属性”的简写形式出现的。当然你也可以明确地罗列出字段名，但在这个例子里面我们让Hibernate来为每个属性注射SQL字段别名。字段别名的占位符是属性名加上表别名的前缀。在下面的例子中，我们从另外一个表（cat_log）中通过映射元数据中的指定获取Cat和它的妈妈。注意，要是我们愿意，我们甚至可以在where子句中使用属性别名。

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mot
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

16.1.4.1. 别名和属性引用(Alias and property references)

大多数情况下，都需要上面的属性注射，但在使用更加复杂的映射，比如复合属性、通过标识符构造继承树，以及集合类等等情况下，也有一些特别的别名，来允许Hibernate注射合适的别名。

下表列出了使用别名注射参数的不同可能性。注意：下面结果中的别名只是示例，实用时每个别名需要唯一并且不同的名字。

表 **16.1.** 别名注射**(alias injection names)**

描述	语法	
简单属性	`{[aliasname].[propertyname]`	`A_N`
复合属性	`{[aliasname].[componentname].[propertyname]}`	`CUR`
实体辨别器 (Discriminator of an entity)	`{[aliasname].class}`	`DIS`
实体的所有属性	`{[aliasname].*}`	`{it`
集合键 (collection key)	`{[aliasname].key}`	`ORG`
集合id	`{[aliasname].id}`	`EMP`
集合元素	`{[aliasname].element}`	`XID`
集合元素的属性	`{[aliasname].element.[propertyname]}`	`NAM`
集合元素的所有属性	`{[aliasname].element.*}`	`{co`
集合的所有属性	`{[aliasname].*}`	`{co`

16.1.5. 返回非受管实体(Returning non-managed entities)

可以对原生sql 查询使用ResultTransformer。这会返回不受Hibernate管理的实体。

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class
```

这个查询指定：

- SQL查询字符串
- 结果转换器(result transformer)

上面的查询将会返回 `CatDTO` 的列表,它将被实例化并且将NAME和BIRTHDAY的值注射入对应的属性或者字段。

16.1.6. 处理继承（Handling inheritance）

原生SQL查询假若其查询结果实体是继承树中的一部分，它必须包含基类和所有子类的所有属性。

16.1.7. 参数（Parameters）

原生查询支持位置参数和命名参数：

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME li
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :na
List pusList = query.setString("name", "Pus%").list();
```

16.2. 命名SQL查询

可以在映射文档中定义查询的名字,然后就可以象调用一个命名的HQL查询一样直接调用命名SQL查询.在这种情况下,我们不 需要调用 `addEntity()` 方法.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

`<return-join>` 和 `<load-collection>` 元素是用来连接关联以及将查询定义为预先初始化各个集合的。

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

一个命名查询可能会返回一个标量值.你必须使用 `<return-scalar>` 元素来指定字段的别名和 Hibernate类型

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

你可以把结果集映射的信息放在外部的 `<resultset>` 元素中，这样就可以在多个命名查询间，或者通过 `setResultSetMapping()` API来访问。(此处原文即存疑。原文为：You can externalize the resultset mapping informations in a `<resultset>` element to either reuse them accross several named queries or through the `setResultSetMapping()` API.)

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

另外,你可以在java代码中直接使用hbm文件中的结果集定义信息。

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten wher
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

16.2.1. 使用return-property来明确地指定字段/别名

使用 `<return-property>` 你可以明确的告诉Hibernate使用哪些字段别名,这取代了使用 `{}` -语法 来让Hibernate注入它自己的别名.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
      <return-property name="name" column="myName"/>
      <return-property name="age" column="myAge"/>
      <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` 也可用于多个字段,它解决了使用 `{}` -语法不能细粒度控制多个字段的限制

```
<sql-query name="organizationCurrentEmployments">
            <return alias="emp" class="Employment">
             <return-property name="salary">
               <return-column name="VALUE"/>
               <return-column name="CURRENCY"/>
             </return-property>
             <return-property name="endDate" column="myEndDate"/>
            </return>
            SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.emp
            STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
            REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE,
            FROM EMPLOYMENT
            WHERE EMPLOYER = :id AND ENDDATE IS NULL
            ORDER BY STARTDATE ASC
</sql-query>
```

注意在这个例子中,我们使用了 `<return-property>` 结合 `{}` 的注入语法. 允许用户来选择如何引用字段以及属性.

如果你映射一个识别器(discriminator),你必须使用 `<return-discriminator>` 来指定识别器字段

16.2.2. 使用存储过程来查询

Hibernate 3引入了对存储过程查询(stored procedure)和函数(function)的支持.以下的说明中，这二者一般都适用。 存储过程/函数必须返回一个结果集,作为Hibernate能够使用的第一个外部参数. 下面是一个Oracle9和更高版本的存储过程例子.

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

在Hibernate里要要使用这个查询,你需要通过命名查询来映射它.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

注意存储过程当前仅仅返回标量和实体.现在不支持 <return-join> 和 <load-collection>

16.2.2.1. 使用存储过程的规则和限制

为了在Hibernate中使用存储过程,你必须遵循一些规则.不遵循这些规则的存储过程将不可用.如果你仍然想要使用他们, 你必须通过 `session.connection()` 来执行他们.这些规则针对于不同的数据库.因为数据库 提供商有各种不同的存储过程语法和语义.

对存储过程进行的查询无法使用 `setFirstResult()/setMaxResults()` 进行分页。

建议采用的调用方式是标准SQL92: `{ ? = call functionName(<parameters>) }` 或者 `{ ? = call procedureName(<parameters>}` .原生调用语法不被支持。

对于Oracle有如下规则:

- 函数必须返回一个结果集。存储过程的第一个参数必须是 `OUT` ，它返回一个结果集。这是通过Oracle 9或10的 `SYS_REFCURSOR` 类型来完成的。在Oracle中你需要定义一个 `REF CURSOR` 类型，参见Oracle的手册。

对于Sybase或者MS SQL server有如下规则:

- 存储过程必须返回一个结果集。.注意这些servers可能返回多个结果集以及更新的数目.Hibernate将取出第一条结果集作为它的返回值， 其他将被丢弃。
- 如果你能够在存储过程里设定 `SET NOCOUNT ON` ，这可能会效率更高，但这不是必需的。

16.3. 定制SQL用来create， update和delete

Hibernate3能够使用定制的SQL语句来执行create,update和delete操作。在Hibernate中，持久化的类和集合已经 包含了一套配置期产生的语句(insertsql, deletesql, updatesql等等)，这些映射标记 <sql-insert> , <sql-delete> , and <sql-update> 重载了 这些语句。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ?
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-upd
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

这些SQL直接在你的数据库里执行，所以你可以自由的使用你喜欢的任意语法。但如果你使用数据库特定的语法， 这当然会降低你映射的可移植性。

如果设定 `callable` ，则能够支持存储过程了。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-ins
    <sql-delete callable="true">{? = call deletePerson (?)}</sql-de
    <sql-update callable="true">{? = call updatePerson (?, ?)}</sql
</class>
```

参数的位置顺序是非常重要的，他们必须和Hibernate所期待的顺序相同。

你能够通过设定日志调试级别为 `org.hiberante.persister.entity` ,来查看Hibernate所期待的顺序。在这个级别下， Hibernate将会打印出create,update和delete实体的静态SQL。(如果想看到预计的顺序。记得不要将定制SQL包含在映射文件里， 因为他们会重载Hibernate生成的静态SQL。)

在大多数情况下(最好这么做)，存储过程需要返回插入/更新/删除的行数，因为Hibernate对语句的成功执行有些运行时的检查。 Hibernate常会把进行CUD操作的语句的第一个参数注册为一个数值型输出参数。

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VA
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

16.4. 定制装载SQL

你可能需要声明你自己的SQL(或HQL)来装载实体

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

这只是一个前面讨论过的命名查询声明，你可以在类映射里引用这个命名查询。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

这也可以用于存储过程

你甚至可以定一个用于集合装载的查询:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

你甚至还可以定义一个实体装载器，它通过连接抓取装载一个集合:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

第 17 章 过滤数据

目录

Hibernate3 提供了一种创新的方式来处理具有“显性(visibility)”规则的数据，那就是使用*Hibernate filter*。 *Hibernate filter*是全局有效的、具有名字、可以带参数的过滤器， 对于某个特定的Hibernate session您可以选择是否启用（或禁用）某个过滤器。

17.1. Hibernate 过滤器(filters)

Hibernate3新增了对某个类或者集合使用预先定义的过滤器条件(filter criteria)的功能。过滤器条件相当于定义一个 非常类似于类和各种集合上的“where”属性的约束子句，但是过滤器条件可以带参数。 应用程序可以在运行时决定是否启用给定的过滤器，以及使用什么样的参数值。 过滤器的用法很像数据库视图，只不过是在应用程序中确定使用什么样的参数的。

要使用过滤器，必须首先在相应的映射节点中定义。而定义一个过滤器，要用到位于 <hibernate-mapping/> 节点之内的 <filter-def/> 节点：

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

定义好之后，就可以在某个类中使用这个过滤器：

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERE
</class>
```

也可以在某个集合使用它：

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERE
</set>
```

可以在多个类或集合中使用某个过滤器；某个类或者集合中也可以使用多个过滤器。

`Session` 对象中会用到的方法有： `enableFilter(String filterName)` , `getEnabledFilter(String filterName)` , 和 `disableFilter(String filterName)` . Session中默认是不启用过滤器的，必须通过 `Session.enabledFilter()` 方法显式的启用。 该方法返回被启用的 `Filter` 的实例。以上文定义的过滤器为例：

```
session.enableFilter("myFilter").setParameter("myFilterParam", "so
```

注意，org.hibernate.Filter的方法允许链式方法调用。（类似上面例子中启用Filter之后设定Filter参数这个“方法链”） Hibernate的其他部分也大多有这个特性。

下面是一个比较完整的例子，使用了记录生效日期模式过滤有时效的数据：

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Departme
    <property name="effectiveStartDate" type="date" column="eff_sta
    <property name="effectiveEndDate" type="date" column="eff_end_c
...
    <!--
        Note that this assumes non-terminal records have an eff_enc
        a max db date for simplicity-sake

        注意，为了简单起见，此处假设雇用关系生效期尚未结束的记录的eff_end_dt
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_c
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_e
    </set>
</class>
```

定义好后，如果想要保证取回的都是目前处于生效期的记录，只需在获取雇员数据的操作之前先开启过滤器即可：

```
Session session = ...;
session.enabledFilter("effectiveDate").setParameter("asOfDate", new
List results = session.createQuery("from Employee as e where e.sala
         .setLong("targetSalary", new Long(1000000))
         .list();
```

在上面的HQL中，虽然我们仅仅显式的使用了一个薪水条件，但因为启用了过滤器，查询将仅返回那些目前雇用 关系处于生效期的，并且薪水高于一百万美刀的雇员的数据。

注意：如果你打算在使用外连接（或者通过HQL或load fetching）的同时使用过滤器，要注意条件表达式的方向（左还是右）。 最安全的方式是使用左外连接（left outer joining）。并且通常来说，先写参数， 然后是操作符，最后写数据库字段名。

在Filter定义之后,它可能被附加到多个实体和/或集合类,每个都有自己的条件。假若这些条件都是一样的，每次都要定义就显得很繁琐。因此， <filter-def/> 被用来定义一个默认条件，它可能作为属性或者CDATA出现：

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

当这个filter被附加到任何目的地，而又没有指明条件时，这个条件就会被使用。注意，换句话说，你可以通过给filter附加特别的条件来重载默认条件。

第 18 章 XML映射

目录

注意这是*Hibernate 3.0*的一个实验性的特性。这一特性仍在积极开发中。

18.1. 用XML数据进行工作

Hibernate使得你可以用XML数据来进行工作，恰如你用持久化的POJO进行工作那样。解析过的XML树 可以被认为是代替POJO的另外一种在对象层面上表示关系型数据的途径.

Hibernate支持采用dom4j作为操作XML树的API。你可以写一些查询从数据库中检索出 dom4j树，随后你对这颗树做的任何修改都将自动同步回数据库。你甚至可以用dom4j解析 一篇XML文档，然后使用Hibernate的任一基本操作将它写入数据库：`persist(), saveOrUpdate(), merge(), delete(), replicate()` (合并操作merge()目前还不支持)。

这一特性可以应用在很多场合，包括数据导入导出，通过JMS或SOAP具体化实体数据以及 基于XSLT的报表。

一个单一的映射就可以将类的属性和XML文档的节点同时映射到数据库。如果不需要映射类， 它也可以用来只映射XML文档。

18.1.1. 指定同时映射XML和类

这是一个同时映射POJO和XML的例子：

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class>
```

18.1.2. 只定义XML映射

这是一个不映射POJO的例子：

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class>
```

这个映射使得你既可以把数据作为一棵dom4j树那样访问，又可以作为由属性键值对(java `Map` s) 组成的图那样访问。属性名字纯粹是逻辑上的结构，你可以在HQL查询中引用它。

18.2. XML映射元数据

许多Hibernate映射元素具有 `node` 属性。这使你可以指定用来保存 属性或实体数据的XML属性或元素。 `node` 属性必须是下列格式之一：

- `"element-name"` - 映射为指定的XML元素
- `"@attribute-name"` - 映射为指定的XML属性
- `"."` - 映射为父元素
- `"element-name/@attribute-name"` - 映射为指定元素的指定属性

对于集合和单值的关联，有一个额外的 `embed-xml` 属性可用。 这个属性的缺省值是真(`embed-xml="true"`)。如果 `embed-xml="true"` ， 则对应于被关联实体或值类型的集合的XML树将直接嵌入拥有这些关联的实体的XML树中。 否则，如果 `embed-xml="false"` ，那么对于单值的关联，仅被引用的实体的标识符出现在XML树中(被引用实体本身不出现)，而集合则根本不出现。

你应该小心，不要让太多关联的embed-xml属性为真(`embed-xml="true"`)，因为XML不能很好地处理 循环引用!

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class>
```

在这个例子中，我们决定嵌入帐目号码(account id)的集合，但不嵌入实际的帐目数据。下面的HQL查询：

```
from Customer c left join fetch c.accounts where c.lastName like :l
```

返回的数据集将是这样：

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings"/>
    <account id="985612323" short-desc="Credit Card"/>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

如果你把一对多映射 `<one-to-many>` 的embed-xml属性置为真 (`embed-xml="true"`), 则数据看上去就像这样：

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance>-2370.34</balance>
    </account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

18.3. 操作XML数据

让我们来读入和更新应用程序中的XML文档。通过获取一个dom4j会话可以做到这一点：

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

将这一特色与Hibernate的 `replicate()` 操作结合起来对于实现的基于XML的数据导入/导出将非常有用.

第 19 章 提升性能

目录

19.1. 抓取策略(Fetching strategies)

抓取策略（*fetching strategy*） 是指：当应用程序需要在（Hibernate实体对象图的）关联关系间进行导航的时候， Hibernate如何获取关联对象的策略。抓取策略可以在O/R映射的元数据中声明，也可以在特定的HQL 或 `条件查询（Criteria Query）` 中重载声明。

Hibernate3 定义了如下几种抓取策略：

- 连接抓取（*Join fetching*） - Hibernate通过 在 `SELECT` 语句使用 `OUTER JOIN` （外连接）来 获得对象的关联实例或者关联集合。
- 查询抓取（*Select fetching*） - 另外发送一条 `SELECT` 语句抓取当前对象的关联实体或集合。除非你显式的指定 `lazy="false"` 禁止 延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条select语句。
- 子查询抓取（*Subselect fetching*） - 另外发送一条 `SELECT` 语句抓取在前面查询到（或者抓取到）的所有实体对象的关联集合。除非你显式的指定 `lazy="false"` 禁止延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条select语句。
- 批量抓取（*Batch fetching*） - 对查询抓取的优化方案， 通过指定一个主键或外键列表，Hibernate使用单条 `SELECT` 语句获取一批对象实例或集合。

Hibernate会区分下列各种情况：

- *Immediate fetching*，立即抓取 - 当宿主被加载时，关联、集合或属性被立即抓取。
- *Lazy collection fetching*，延迟集合抓取- 直到应用程序对集合进行了一次操作时，集合才被抓取。（对集合而言这是默认行为。）
- *"Extra-lazy" collection fetching,"Extra-lazy"*集合抓取 -对集合类中的每个元素而言，都是直到需要时才去访问数据库。除非绝对必要，Hibernate不会试图去把整个集合都抓取到内存里来（适用于非常大的集合）。
- *Proxy fetching*，代理抓取 - 对返回单值的关联而言，当其某个方法被调用，而非对其关键字进行get操作时才抓取。
- *"No-proxy" fetching,*非代理抓取 - 对返回单值的关联而言，当实例变量被访问的时候进行抓取。与上面的代理抓取相比，这种方法没有那么“延迟”得厉害(就算只访问标识符，也会导致关联抓取)但是更加透明，因为对应用程序来说，不再看到proxy。这种方法需要在编译期间进行字节码增强操作，因此很少需要用到。
- *Lazy attribute fetching*，属性延迟加载 - 对属性或返回单值的关联而言，当其实例变量被访问的时候进行抓取。需要编译期字节码强化，因此这一方法很少是必要的。

这里有两个正交的概念：关联何时被抓取，以及被如何抓取（会采用什么样的SQL语句）。不要混淆它们！我们使用 `抓取` 来改善性能。我们使用 `延迟` 来定义一些契约，对某特定类的某个脱管的实例，知道有哪些数据是可以使用的。

19.1.1. 操作延迟加载的关联

默认情况下，Hibernate 3对集合使用延迟select抓取，对返回单值的关联使用延迟代理抓取。对几乎是所有的应用而言，其绝大多数的关联，这种策略都是有效的。

注意:假若你设置了 `hibernate.default_batch_fetch_size` ,Hibernate会对延迟加载采取批量抓取优化措施（这种优化也可能会在更细化的级别打开）。

然而，你必须了解延迟抓取带来的一个问题。在一个打开的Hibernate session上下文之外调用延迟集合会导致一次意外。比如：

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Er
```

在 `Session` 关闭后，permessions集合将是未实例化的、不再可用，因此无法正常载入其状态。 *Hibernate*对脱管对象不支持延迟实例化. 这里的修改方法是：将permissions读取数据的代码 移到tx.commit()之前。

除此之外，通过对关联映射指定 `lazy="false"` ,我们也可以使用非延迟的集合或关联。但是， 对绝大部分集合来说，更推荐使用延迟方式抓取数据。如果在你的对象模型中定义了太多的非延迟关联，Hibernate最终几乎需要在每个事务中载入整个数据库到内存中！

但是，另一方面，在一些特殊的事务中，我们也经常需要使用到连接抓取（它本身上就是非延迟的），以代替查询抓取。 下面我们将会很快明白如何具体的定制Hibernate中的抓取策略。在Hibernate3中，具体选择哪种抓取策略的机制是和选择单值关联或集合关联相一致的。

19.1.2. 调整抓取策略（Tuning fetch strategies）

查询抓取（默认的）在N+1查询的情况下是极其脆弱的，因此我们可能会要求在映射文档中定义使用连接抓取：

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

在映射文档中定义的 `抓取` 策略将会对以下列表条目产生影响：

- 通过 `get()` 或 `load()` 方法取得数据。
- 只有在关联之间进行导航时，才会隐式的取得数据。
- `条件查询`
- 使用了 `subselect` 抓取的HQL查询

不管你使用哪种抓取策略，定义为非延迟的类图会被保证一定装载入内存。注意这可能意味着在一条HQL查询后紧跟着一系列的查询。

通常情况下，我们并不使用映射文档进行抓取策略的定制。更多的是，保持其默认值，然后在特定的事务中， 使用HQL的 `左连接抓取（left join fetch）` 对其进行重载。这将通知 Hibernate在第一次查询中使用外部关联（outer join），直接得到其关联数据。 在 `条件查询` API中，应该调用 `setFetchMode(FetchMode.JOIN)` 语句。

也许你喜欢仅仅通过条件查询，就可以改变 `get()` 或 `load()` 语句中的数据抓取策略。例如：

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

（这就是其他ORM解决方案的“抓取计划(fetch plan)”在Hibernate中的等价物。）

截然不同的一种避免N+1次查询的方法是，使用二级缓存。

19.1.3. 单端关联代理（Single-ended association proxies）

在Hinerbate中，对集合的延迟抓取的采用了自己的实现方法。但是，对于单端关联的延迟抓取，则需要采用 其他不同的机制。单端关联的目标实体必须使用代理，Hihernate在运行期二进制级（通过优异的CGLIB库）， 为持久对象实现了延迟载入代理。

默认的，Hibernate3将会为所有的持久对象产生代理（在启动阶段），然后使用他们实现 `多对一（many-to-one）` 关联和 `一对一（one-to-one）` 关联的延迟抓取。

在映射文件中，可以通过设置 `proxy` 属性为目标class声明一个接口供代理接口使用。 默认的，Hibernate将会使用该类的一个子类。 注意：被代理的类必须实现一个至少包可见的默认构造函数，我们建议所有的持久类都应拥有这样的构造函数

在如此方式定义一个多态类的时候，有许多值得注意的常见性的问题，例如：

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

首先， `Cat` 实例永远不可以被强制转换为 `DomesticCat` , 即使它本身就是 `DomesticCat` 实例。

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a prox
if ( cat.isDomesticCat() ) {                  // hit the db to init
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

其次，代理的“ `==` ”可能不再成立。

```
Cat cat = (Cat) session.load(Cat.class, id);            // instant:
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // acqu
System.out.println(cat==dc);                            // false
```

虽然如此，但实际情况并没有看上去那么糟糕。虽然我们现在有两个不同的引用，分别指向这两个不同的代理对象， 但实际上，其底层应该是同一个实例对象：

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

第三，你不能对“final类”或“具有final方法的类”使用CGLIB代理。

最后，如果你的持久化对象在实例化时需要某些资源（例如，在实例化方法、默认构造方法中）， 那么代理对象也同样需要使用这些资源。实际上，代理类是持久化类的子类。

这些问题都源于Java的单根继承模型的天生限制。如果你希望避免这些问题，那么你的每个持久化类必须实现一个接口， 在此接口中已经声明了其业务方法。然后，你需要在映射文档中再指定这些接口。例如：

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

这里 `CatImpl` 实现了 `Cat` 接口， `DomesticCatImpl` 实现 `DomesticCat` 接口。 在 `load()` 、 `iterate()` 方法中就会返回 `Cat` 和 `DomesticCat` 的代理对象。 (注意 `list()` 并不会返回代理对象。)

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.iterate("from CatImpl as cat where cat.name
Cat fritz = (Cat) iter.next();
```

这里，对象之间的关系也将被延迟载入。这就意味着，你应该将属性声明为 `Cat` ，而不是 `CatImpl` 。

但是，在有些方法中是不需要使用代理的。例如：

- `equals()` 方法，如果持久类没有重载 `equals()` 方法。
- `hashCode()` 方法，如果持久类没有重载 `hashCode()` 方法。
- 标志符的getter方法。

Hibernate将会识别出那些重载了 `equals()` 、或 `hashCode()` 方法的持久化类。

若选择 `lazy="no-proxy"` 而非默认的 `lazy="proxy"` ，我们可以避免类型转换带来的问题。然而，这样我们就需要编译期字节码增强，并且所有的操作都会导致立刻进行代理初始化。

19.1.4. 实例化集合和代理（Initializing collections and proxies）

在 `Session` 范围之外访问未初始化的集合或代理，Hibernate将会抛出 `LazyInitializationException` 异常。 也就是说，在分离状态下，访问一个实体所拥有的集合，或者访问其指向代理的属性时，会引发此异常。

有时候我们需要保证某个代理或者集合在Session关闭前就已经被初始化了。 当然，我们可以通过强行调用 `cat.getSex()` 或者 `cat.getKittens().size()` 之类的方法来确保这一点。 但是这样的程序会造成读者的疑惑，也不符合通常的代码规范。

静态方法 `Hibernate.initialized()` 为你的应用程序提供了一个便捷的途径来延迟加载集合或代理。 只要它的Session处于open状态， `Hibernate.initialize(cat)` 将会为cat强制对代理实例化。 同样， `Hibernate.initialize( cat.getKittens() )` 对kittens的集合具有同样的功能。

还有另外一种选择，就是保持 `Session` 一直处于open状态，直到所有需要的集合或代理都被载入。 在某些应用架构中，特别是对于那些使用Hibernate进行数据访问的代码，以及那些在不同应用层和不同物理进程中使用Hibernate的代码。 在集合实例化时，如何保证 `Session` 处于open状态经常会是一个问题。有两种方法可以解决此问题：

- 在一个基于Web的应用中，可以利用servlet过滤器（filter），在用户请求（request）结束、页面生成 结束时关闭 `Session` （这里使用了在展示层保持打开*Session*模式（*Open Session in View*））， 当然，这将依赖于应用框架中异常需要被正确的处理。在返回界面给用户之前，乃至在生成界面过程中发生异常的情况下， 正确关闭 `Session` 和结束事务将是非常重要的， 请参见Hibernate wiki上的"Open Session in View"模式，你可以找到示例。

- 在一个拥有单独业务层的应用中，业务层必须在返回之前，为web层“准备”好其所需的数据集合。这就意味着 业务层应该载入所有表现层/web层所需的数据，并将这些已实例化完毕的数据返回。通常，应用程序应该 为web层所需的每个集合调用 `Hibernate.initialize()` （这个调用必须发生咱session关闭之前）；或者使用带有 `FETCH` 从句，或 `FetchMode.JOIN` 的Hibernate查询， 事先取得所有的数据集合。如果你在应用中使用了*Command*模式，代替*Session Facade* ， 那么这项任务将会变得简单的多。

- 你也可以通过 `merge()` 或 `lock()` 方法，在访问未实例化的集合（或代理）之前， 为先前载入的对象绑定一个新的 `Session` 。 显然，Hibernate将不会，也不应该自动完成这些任务，因为这将引入一个特殊的事务语义。

有时候，你并不需要完全实例化整个大的集合，仅需要了解它的部分信息（例如其大小）、或者集合的部分内容。

你可以使用集合过滤器得到其集合的大小，而不必实例化整个集合：

```
( (Integer) s.createFilter( collection, "select count(*)" ).list()
```

这里的 `createFilter()` 方法也可以被用来有效的抓取集合的部分内容，而无需实例化整个集合：

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults
```

19.1.5. 使用批量抓取（Using batch fetching）

Hibernate可以充分有效的使用批量抓取，也就是说，如果仅一个访问代理（或集合），那么Hibernate将不载入其他未实例化的代理。 批量抓取是延迟查询抓取的优化方案，你可以在两种批量抓取方案之间进行选择：在类级别和集合级别。

类/实体级别的批量抓取很容易理解。假设你在运行时将需要面对下面的问题：你在一个 `Session` 中载入了25个 `Cat` 实例，每个 `Cat` 实例都拥有一个引用成员 `owner` ， 其指向 `Person` ，而 `Person` 类是代理，同时 `lazy="true"` 。 如果你必须遍历整个cats集合，对每个元素调用 `getOwner()` 方法，Hibernate将会默认的执行25次 `SELECT` 查询， 得到其owner的代理对象。这时，你可以通过在映射文件的 `Person` 属性，显式声明 `batch-size` ，改变其行为：

```
<class name="Person" batch-size="10">...</class>
```

随之，Hibernate将只需要执行三次查询，分别为10、10、 5。

你也可以在集合级别定义批量抓取。例如，如果每个 `Person` 都拥有一个延迟载入的 `Cats` 集合， 现在， `Sesssion` 中载入了10个person对象，遍历person集合将会引起10次 `SELECT` 查询， 每次查询都会调用 `getCats()` 方法。如果你在 `Person` 的映射定义部分，允许对 `cats` 批量抓取, 那么，Hibernate将可以预先抓取整个集合。请看例子：

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

如果整个的 `batch-size` 是3（笔误？），那么Hibernate将会分四次执行 `SELECT` 查询， 按照3、3、3、1的大小分别载入数据。这里的每次载入的数据量还具体依赖于当前 `Session` 中未实例化集合的个数。

如果你的模型中有嵌套的树状结构，例如典型的帐单－原料结构（bill-of-materials pattern），集合的批量抓取是非常有用的。 （尽管在更多情况下对树进行读取时，嵌套集合（*nested set*）或原料路径*(materialized path)*（××） 是更好的解决方法。）

19.1.6. 使用子查询抓取（Using subselect fetching）

假若一个延迟集合或单值代理需要抓取，Hibernate会使用一个subselect重新运行原来的查询，一次性读入所有的实例。这和批量抓取的实现方法是一样的，不会有破碎的加载。

19.1.7. 使用延迟属性抓取（Using lazy property fetching）

Hibernate3对单独的属性支持延迟抓取，这项优化技术也被称为组抓取（*fetch groups*）。 请注意，该技术更多的属于市场特性。在实际应用中，优化行读取比优化列读取更重要。但是，仅载入类的部分属性在某些特定情况下会有用，例如在原有表中拥有几百列数据、数据模型无法改动的情况下。

可以在映射文件中对特定的属性设置 `lazy` ，定义该属性为延迟载入。

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="tru
    <property name="text" not-null="true" length="2000" lazy="true'
</class>
```

属性的延迟载入要求在其代码构建时加入二进制指示指令（bytecode instrumentation），如果你的持久类代码中未含有这些指令， Hibernate将会忽略这些属性的延迟设置，仍然将其直接载入。

你可以在Ant的Task中，进行如下定义，对持久类代码加入“二进制指令。”

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instru
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/mode
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

还有一种可以优化的方法，它使用HQL或条件查询的投影（projection）特性，可以避免读取非必要的列， 这一点至少对只读事务是非常有用的。它无需在代码构建时“二进制指令”处理，因此是一个更加值得选择的解决方法。

有时你需要在HQL中通过 `抓取所有属性` ，强行抓取所有内容。

19.2. 二级缓存（The Second Level Cache）

Hibernate的 `Session` 在事务级别进行持久化数据的缓存操作。 当然，也有可能分别为每个类（或集合)，配置集群、或JVM级别(`SessionFactory级别`)的缓存。你甚至可以为之插入一个集群的缓存。注意，缓存永远不知道其他应用程序对持久化仓库（数据库）可能进行的修改 （即使可以将缓存数据设定为定期失效）。

通过在 `hibernate.cache.provider_class` 属性中指定 `org.hibernate.cache.CacheProvider` 的某个实现的类名,你可以选择让Hibernate使用哪个缓存实现。Hibernate打包一些开源缓存实现，提供对它们的内置支持（见下表）。除此之外，你也可以实现你自己的实现，将它们插入到系统中。注意，在3.2版本之前，默认使用EhCache 作为缓存实现，但从3.2起就不再这样了。

表 **19.1.** 缓存策略提供商（**Cache Providers**）

Cache	Provider class	Typ
Hashtable (not intended for production use)	`org.hibernate.cache.HashtableCacheProvider`	memor
EHCache	`org.hibernate.cache.EhCacheProvider`	memor disk
OSCache	`org.hibernate.cache.OSCacheProvider`	memor disk
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	clustere multica
JBoss TreeCache	`org.hibernate.cache.TreeCacheProvider`	clustere multica transac

19.2.1. 缓存映射（Cache mappings）

类或者集合映射的“ `<cache>` 元素”可以有下列形式：

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"
    region="RegionName"
    include="all|non-lazy"
/>
```

❶	**`usage`** **(必须)说明了缓存的策略:** **`transactional`** 、 **`read-write`** 、 **`nonstrict-read-write`** 或 **`read-only`** 。
❷	`region` (可选，默认为类或者集合的名字(class or collection role name)) 指定第二级缓存的区域名(name of the second level cache region)
❸	`include` (可选,默认为 `all`) `non-lazy` 当属性级延迟抓取打开时，标记为 `lazy="true"` 的实体的属性可能无法被缓存

另外(首选?), 你可以在hibernate.cfg.xml中指定 `<class-cache>` 和 `<collection-cache>` 元素。

这里的 `usage` 属性指明了缓存并发策略（*cache concurrency strategy*）。

19.2.2. 策略：只读缓存（Strategy: read only）

如果你的应用程序只需读取一个持久化类的实例，而无需对其修改， 那么就可以对其进行 `只读` 缓存。这是最简单，也是实用性最好的方法。甚至在集群中，它也能完美地运作。

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

19.2.3. 策略:读/写缓存（Strategy: read/write）

如果应用程序需要更新数据，那么使用`读/写缓存` 比较合适。 如果应用程序要求“序列化事务”的隔离级别（serializable transaction isolation level），那么就决不能使用这种缓存策略。 如果在JTA环境中使用缓存，你必须指定 `hibernate.transaction.manager_lookup_class` 属性的值， 通过它，Hibernate才能知道该应用程序中JTA的 `TransactionManager` 的具体策略。 在其它环境中，你必须保证在 `Session.close()` 、或 `Session.disconnect()` 调用前， 整个事务已经结束。 如果你想在集群环境中使用此策略，你必须保证底层的缓存实现支持锁定(locking)。Hibernate内置的缓存策略并不支持锁定功能。

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

19.2.4. 策略:非严格读/写缓存（Strategy: nonstrict read/write）

如果应用程序只偶尔需要更新数据（也就是说，两个事务同时更新同一记录的情况很不常见），也不需要十分严格的事务隔离， 那么比较适合使用 `非严格读/写缓存` 策略。如果在JTA环境中使用该策略， 你必须为其指定 `hibernate.transaction.manager_lookup_class` 属性的值， 在其它环境中，你必须保证在 `Session.close()` 、或 `Session.disconnect()` 调用前， 整个事务已经结束。

19.2.5. 策略:事务缓存（transactional）

Hibernate的 `事务缓存` 策略提供了全事务的缓存支持， 例如对JBoss TreeCache的支持。这样的缓存只能用于JTA环境中，你必须指定 为其 `hibernate.transaction.manager_lookup_class` 属性。

没有一种缓存提供商能够支持上列的所有缓存并发策略。下表中列出了各种提供器、及其各自适用的并发策略。

表 19.2. 各种缓存提供商对缓存并发策略的支持情况（Cache Concurrency Strategy Support）

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes	yes		

19.3. 管理缓存（Managing the caches）

无论何时，当你给 `save()` 、 `update()` 或 `saveOrUpdate()` 方法传递一个对象时，或使用 `load()` 、 `get()` 、 `list()` 、 `iterate()` 或 `scroll()` 方法获得一个对象时, 该对象都将被加入到 `Session` 的内部缓存中。

当随后flush()方法被调用时，对象的状态会和数据库取得同步。 如果你不希望此同步操作发生，或者你正处理大量对象、需要对有效管理内存时，你可以调用 `evict()` 方法，从一级缓存中去掉这些对象及其集合。

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

Session还提供了一个 `contains()` 方法，用来判断某个实例是否处于当前session的缓存中。

如若要把所有的对象从session缓存中彻底清除，则需要调用 `Session.clear()` 。

对于二级缓存来说，在 `SessionFactory` 中定义了许多方法， 清除缓存中实例、整个类、集合实例或者整个集合。

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a par
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten c
```

`CacheMode` 参数用于控制具体的Session如何与二级缓存进行交互。

- `CacheMode.NORMAL` - 从二级缓存中读、写数据。
- `CacheMode.GET` - 从二级缓存中读取数据，仅在数据更新时对二级缓存写数据。
- `CacheMode.PUT` - 仅向二级缓存写数据，但不从二级缓存中读数据。
- `CacheMode.REFRESH` - 仅向二级缓存写数据，但不从二级缓存中读数据。通过 `hibernate.cache.use_minimal_puts` 的设置，强制二级缓存从数据库中读取数据，刷新缓存内容。

如若需要查看二级缓存或查询缓存区域的内容，你可以使用`统计（Statistics）` API。

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

此时，你必须手工打开统计选项。可选的，你可以让Hibernate更人工可读的方式维护缓存内容。

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

19.4. 查询缓存（The Query Cache）

查询的结果集也可以被缓存。只有当经常使用同样的参数进行查询时，这才会有些用处。 要使用查询缓存，首先你必须打开它：

```
hibernate.cache.use_query_cache true
```

该设置将会创建两个缓存区域 - 一个用于保存查询结果集(`org.hibernate.cache.StandardQueryCache`)； 另一个则用于保存最近查询的一系列表的时间戳(`org.hibernate.cache.UpdateTimestampsCache`)。 请注意：在查询缓存中，它并不缓存结果集中所包含的实体的确切状态；它只缓存这些实体的标识符属性的值、以及各值类型的结果。 所以查询缓存通常会和二级缓存一起使用。

绝大多数的查询并不能从查询缓存中受益，所以Hibernate默认是不进行查询缓存的。如若需要进行缓存，请调用 `Query.setCacheable(true)` 方法。这个调用会让查询在执行过程中时先从缓存中查找结果， 并将自己的结果集放到缓存中去。

如果你要对查询缓存的失效政策进行精确的控制，你必须调用 `Query.setCacheRegion()` 方法， 为每个查询指定其命名的缓存区域。

```
List blogs = sess.createQuery("from Blog blog where blog.blogger =
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

如果查询需要强行刷新其查询缓存区域，那么你应该调用 `Query.setCacheMode(CacheMode.REFRESH)` 方法。 这对在其他进程中修改底层数据（例如，不通过Hibernate修改数据），或对那些需要选择性更新特定查询结果集的情况特别有用。 这是对 `SessionFactory.evictQueries()` 的更为有效的替代方案，同样可以清除查询缓存区域。

19.5. 理解集合性能（Understanding Collection performance）

前面我们已经对集合进行了足够的讨论。本段中，我们将着重讲述集合在运行时的事宜。

19.5.1. 分类（Taxonomy）

Hibernate定义了三种基本类型的集合：

- 值数据集合
- 一对多关联
- 多对多关联

这个分类是区分了不同的表和外键关系类型，但是它没有告诉我们关系模型的所有内容。 要完全理解他们的关系结构和性能特点，我们必须同时考虑“用于Hibernate更新或删除集合行数据的主键的结构”。 因此得到了如下的分类：

- 有序集合类
- 集合（sets）
- 包（bags）

所有的有序集合类（maps, lists, arrays)都拥有一个由 `<key>` 和 `<index>` 组成的主键。 这种情况下集合类的更新是非常高效的——主键已经被有效的索引，因此当Hibernate试图更新或删除一行时，可以迅速找到该行数据。

集合(sets)的主键由 `<key>` 和其他元素字段构成。 对于有些元素类型来说，这很低效，特别是组合元素或者大文本、大二进制字段；数据库可能无法有效的对复杂的主键进行索引。 另一方面，对于一对多、多对多关联，特别是合成的标识符来说，集合也可以达到同样的高效性能。（附注：如果你希望 `SchemaExport` 为你的 `<set>` 创建主键， 你必须把所有的字段都声明为 `not-null="true"` 。）

`<idbag>` 映射定义了代理键，因此它总是可以很高效的被更新。事实上， `<idbag>` 拥有着最好的性能表现。

Bag是最差的。因为bag允许重复的元素值，也没有索引字段，因此不可能定义主键。 Hibernate无法判断出重复的行。当这种集合被更改时，Hibernate将会先完整地移除 （通过一个(in a single `DELETE`)）整个集合，然后再重新创建整个集合。因此Bag是非常低效的。

请注意：对于一对多关联来说，“主键”很可能并不是数据库表的物理主键。 但就算在此情况下，上面的分类仍然是有用的。（它仍然反映了Hibernate在集合的各数据行中是如何进行“定位”的。）

19.5.2. Lists, maps 和sets用于更新效率最高

根据我们上面的讨论，显然有序集合类型和大多数set都可以在增加、删除、修改元素中拥有最好的性能。

可论证的是对于多对多关联、值数据集合而言，有序集合类比集合(set)有一个好处。因为 `Set` 的内在结构， 如果“改变”了一个元素，Hibernate并不会 `更新（UPDATE）` 这一行。 对于 `Set` 来说，只有在 `插入（INSERT）` 和 `删除（DELETE）` 操作时“改变”才有效。再次强调：这段讨论对“一对多关联”并不适用。

注意到数组无法延迟载入，我们可以得出结论，list, map和idbags是最高效的（非反向）集合类型，set则紧随其后。 在Hibernate中，set应该时最通用的集合类型，这时因为“set”的语义在关系模型中是最自然的。

但是，在设计良好的Hibernate领域模型中，我们通常可以看到更多的集合事实上是带有 `inverse="true"` 的一对多的关联。对于这些关联，更新操作将会在多对一的这一端进行处理。因此对于此类情况，无需考虑其集合的更新性能。

19.5.3. Bag和list是反向集合类中效率最高的

在把bag扔进水沟之前，你必须了解，在一种情况下，bag的性能(包括list)要比set高得多： 对于指明了 `inverse="true"` 的集合类（比如说，标准的双向的一对多关联）， 我们可以在未初始化(fetch)包元素的情况下直接向bag或list添加新元素！ 这是因为 `Collection.add()`)或者 `Collection.addAll()` 方法 对bag或者List总是返回true（这点与与Set不同）。因此对于下面的相同代码来说，速度会快得多。

```
Parent p = (Parent) sess.load(Parent.class, id);
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c);  //no need to fetch the collection!
    sess.flush();
```

19.5.4. 一次性删除（One shot delete）

偶尔的，逐个删除集合类中的元素是相当低效的。Hibernate并没那么笨， 如果你想要把整个集合都删除（比如说调用list.clear()），Hibernate只需要一个DELETE就搞定了。

假设我们在一个长度为20的集合类中新增加了一个元素，然后再删除两个。Hibernate会安排一条 `INSERT` 语句和两条 `DELETE` 语句（除非集合类是一个bag)。 这当然是显而易见的。

但是，假设我们删除了18个数据，只剩下2个，然后新增3个。则有两种处理方式：

- 逐一的删除这18个数据，再新增三个；
- 删除整个集合类（只用一句DELETE语句），然后增加5个数据。

Hibernate还没那么聪明，知道第二种选择可能会比较快。 （也许让Hibernate不这么聪明也是好事，否则可能会引发意外的“数据库触发器”之类的问题。）

幸运的是，你可以强制使用第二种策略。你需要取消原来的整个集合类（解除其引用）， 然后再返回一个新的实例化的集合类，只包含需要的元素。有些时候这是非常有用的。

显然，一次性删除并不适用于被映射为 `inverse="true"` 的集合。

19.6. 监测性能（Monitoring performance）

没有监测和性能参数而进行优化是毫无意义的。Hibernate为其内部操作提供了一系列的示意图，因此可以从 每个 `SessionFactory` 抓取其统计数据。

19.6.1. 监测SessionFactory

你可以有两种方式访问 `SessionFactory` 的数据记录，第一种就是自己直接调用 `sessionFactory.getStatistics()` 方法读取、显示 `统计` 数据。

此外，如果你打开 `StatisticsService` MBean选项，那么Hibernate则可以使用JMX技术 发布其数据记录。你可以让应用中所有的 `SessionFactory` 同时共享一个MBean，也可以每个 SessionFactory分配一个MBean。下面的代码即是其演示代码：

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object na

StatisticsService stats = new StatisticsService(); // MBean impleme
stats.setSessionFactory(sessionFactory); // Bind the stats to a Ses
server.registerMBean(stats, on); // Register the Mbean on the serve
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object na

StatisticsService stats = new StatisticsService(); // MBean impleme
server.registerMBean(stats, on); // Register the MBean on the serve
```

TODO：仍需要说明的是：在第一个例子中，我们直接得到和使用MBean；而在第二个例子中，在使用MBean之前 我们则需要给出SessionFactory的JNDI名，使用 `hibernateStatsBean.setSessionFactoryJNDIName("my/JNDI/Name")` 得到SessionFactory，然后将MBean保存于其中。

你可以通过以下方法打开或关闭 `SessionFactory` 的监测功能：

- 在配置期间，将 `hibernate.generate_statistics` 设置为 `true` 或 `false` ；
- 在运行期间，则可以可以通过 `sf.getStatistics().setStatisticsEnabled(true)` 或 `hibernateStatsBean.setStatisticsEnabled(true)`

你也可以在程序中调用 `clear()` 方法重置统计数据，调用 `logSummary()` 在日志中记录（info级别）其总结。

19.6.2. 数据记录（Metrics）

Hibernate提供了一系列数据记录，其记录的内容包括从最基本的信息到与具体场景的特殊信息。所有的测量值都可以由 `Statistics` 接口进行访问，主要分为三类：

- 使用 `Session` 的普通数据记录，例如打开的Session的个数、取得的JDBC的连接数等；
- 实体、集合、查询、缓存等内容的统一数据记录
- 和具体实体、集合、查询、缓存相关的详细数据记录

例如：你可以检查缓存的命中成功次数，缓存的命中失败次数，实体、集合和查询的使用概率，查询的平均时间等。请注意 Java中时间的近似精度是毫秒。Hibernate的数据精度和具体的JVM有关，在有些平台上其精度甚至只能精确到10秒。

你可以直接使用getter方法得到全局数据记录（例如，和具体的实体、集合、缓存区无关的数据），你也可以在具体查询中通过标记实体名、 或HQL、SQL语句得到某实体的数据记录。请参考 `Statistics` 、 `EntityStatistics` 、 `CollectionStatistics` 、 `SecondLevelCacheStatistics` 、和 `QueryStatistics` 的API文档以抓取更多信息。下面的代码则是个简单的例子：

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

如果你想得到所有实体、集合、查询和缓存区的数据，你可以通过以下方法获得实体、集合、查询和缓存区列表： `getQueries()` 、 `getEntityNames()` 、 `getCollectionRoleNames()` 和 `getSecondLevelCacheRegionNames()` 。

第 20 章 工具箱指南

目录

可以通过一系列Eclipse插件、命令行工具和Ant任务来进行与Hibernate关联的转换。

除了Ant任务外，当前的*Hibernate Tools*也包含了Eclipse IDE的插件，用于与现存数据库的逆向工程。

- *Mapping Editor:* Hibernate XML映射文件的编辑器，支持自动完成和语法高亮。它也支持对类名和属性/字段名的语义自动完成，比通常的XML编辑器方便得多。

- *Console:* Console是Eclipse的一个新视图。除了对你的console配置的树状概览，你还可以获得对你持久化类及其关联的交互式视图。Console允许你对数据库执行HQL查询，并直接在Eclipse中浏览结果。

- *Development Wizards:* 在Hibernate Eclipse tools中还提供了几个向导；你可以用向导快速生成Hibernate 配置文件（cfg.xml），你甚至还可以同现存的数据库schema中反向工程出POJO源代码与Hibernate 映射文件。反向工程支持可定制的模版。

- *Ant Tasks:*

要得到更多信息，请查阅 *Hibernate Tools* 包及其文档。

同时，Hibernate主发行包还附带了一个集成的工具（它甚至可以在Hibernate“内部”快速运行）*SchemaExport*，也就是 `hbm2ddl` 。

20.1. Schema自动生成（Automatic schema generation）

可以从你的映射文件使用一个Hibernate工具生成DDL。 生成的schema包含有对实体和集合类表的完整性引用约束（主键和外键）。涉及到的标示符生成器所需的表和sequence也会同时生成。

在使用这个工具的时候，你必须 通过 `hibernate.dialet` 属性指定一个SQL `方言(Dialet)` ，因为DDL是与供应商高度相关的。

首先，要定制你的映射文件，来改善生成的schema。

20.1.1. 对schema定制化(Customizing the schema)

很多Hibernate映射元素定义了可选的 `length` 、 `precision` 或者 `scale` 属性。你可以通过这个属性设置字段的长度、精度、小数点位数。

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

有些tag还接受 `not-null` 属性（用来在表字段上生成 `NOT NULL` 约束）和 `unique` 属性（用来在表字段上生成 `UNIQUE` 约束）。

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique='
```

`unique-key` 属性可以对成组的字段指定一个唯一键约束(unique key constraint)。目前， `unique-key` 属性指定的值在生成DDL时并不会被当作这个约束的名字，它们只是在用来在映射文件内部用作区分的。

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

`index` 属性会用对应的字段（一个或多个）生成一个index,它指出了这个index的名字。如果多个字段对应的index名字相同，就会生成包含这些字段的index。

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

`foreign-key` 属性可以用来覆盖任何生成的外键约束的名字。

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

很多映射元素还接受 <column> 子元素。这在定义跨越多字段的类型时特别有用。

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30
    <column name="first" not-null="true" index="bar_idx" length="2
    <column name="initial"/>
</property>
```

`default` 属性为字段指定一个默认值 (在保存被映射的类的新实例之前，你应该将同样的值赋于对应的属性)。

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property>
```

`sql-type` 属性允许用户覆盖默认的Hibernate类型到SQL数据类型的映射。

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

`check` 属性允许用户指定一个约束检查。

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

表 **20.1. Summary**

属性(Attribute)	值（Values）	解释（Interpretatior
`length`	数字	字段长度
`precision`	数字	精度(decimal precision)
`scale`	数字	小数点位数(decimal scale)
`not-null`	`true|false`	指明字段是否应该是非空的
`unique`	`true|false`	指明是否该字段具有惟一约束
`index`	`index_name`	指明一个（多字段）的索引(index)的
`unique-key`	`unique_key_name`	指明多字段惟一约束的名字（参见上
`foreign-key`	`foreign_key_name`	specifies the name of the foreign key generated for an association, for a `<one-to-one>` , `<many` `<key>` , or `<many-to-m` mapping element. Note that `invers` will not be considered by `SchemaEx` 外键的名字，它是为关联生成的，或者 `<one-to-one>` ， `<ma` `<key>` ，或者 `<many-to-` 元素。注意 `inverse="true"` 在 `Sc` 被忽略。
`sql-type`	`SQL 字段类型`	覆盖默认的字段类型(只能用于 `<` 性）
`default`	SQL表达式	为字段指定默认值
`check`	SQL 表达式	对字段或表加入SQL约束检查

`<comment>` 元素可以让你在生成的schema中加入注释。

```
<class name="Customer" table="CurCust">
    <comment>Current customers only</comment>
    ...
</class>
```

```
<property name="balance">
    <column name="bal">
        <comment>Balance in USD</comment>
    </column>
</property>
```

结果是在生成的DDL中包含 `comment on table` 或者 `comment on column` 语句(假若支持的话)。

20.1.2. 运行该工具

`SchemaExport` 工具把DDL脚本写到标准输出，同时/或者执行DDL语句。

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

表 20.2. `SchemaExport` 命令行选项

选项	说明
`--quiet`	不要把脚本输出到stdout
`--drop`	只进行drop tables的步骤
`--create`	只创建表
`--text`	不执行在数据库中运行的步骤
`--output=my_schema.ddl`	把输出的ddl脚本输出到一个文件
`--naming=eg.MyNamingStrategy`	选择一个命名策略（ `NamingStrategy` ）
`--config=hibernate.cfg.xml`	从XML文件读入Hibernate配置
`--properties=hibernate.properties`	从文件读入数据库属性
`--format`	把脚本中的SQL语句对齐和美化
`--delimiter=;`	为脚本设置行结束符

你甚至可以在你的应用程序中嵌入 `SchemaExport` 工具:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

20.1.3. 属性(Properties)

可以通过如下方式指定数据库属性:

- 通过 `-D` *<property>*系统参数
- 在 `hibernate.properties` 文件中
- 位于一个其它名字的properties文件中,然后用 `--properties` 参数指定

所需的参数包括:

表 **20.3. SchemaExport** 连接属性

属性名	说明
`hibernate.connection.driver_class`	jdbc driver class
`hibernate.connection.url`	jdbc url
`hibernate.connection.username`	database user
`hibernate.connection.password`	user password
`hibernate.dialect`	方言(dialect)

20.1.4. 使用Ant(Using Ant)

你可以在你的Ant build脚本中调用 `SchemaExport` :

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

20.1.5. 对schema的增量更新(Incremental schema updates)

`SchemaUpdate` 工具对已存在的schema采用"增量"方式进行更新。注意 `SchemaUpdate` 严重依赖于JDBC metadata API,所以它并非对所有JDBC驱动都有效。

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

表 20.4. **`SchemaUpdate`** 命令行选项

选项	说明
`--quiet`	不要把脚本输出到stdout
`--text`	不把脚本输出到数据库
`--naming=eg.MyNamingStrategy`	选择一个命名策略 (`NamingStrategy`)
`--properties=hibernate.properties`	从指定文件读入数据库属性
`--config=hibernate.cfg.xml`	指定一个 `.cfg.xml` 文件

你可以在你的应用程序中嵌入 `SchemaUpdate` 工具:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

20.1.6. 用Ant来增量更新schema(Using Ant for incremental schema updates)

你可以在Ant脚本中调用 `SchemaUpdate` ：

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

20.1.7. Schema 校验

`SchemaValidator` 工具会比较数据库现状是否与映射文档“匹配”。注意，`SchemaValidator` 严重依赖于JDBC的metadata API，因此不是对所有的JDBC驱动都适用。这一工具在测试的时候特别有用。

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

表 **20.5.** **`SchemaValidator`** 命令行参数

选项	描述
`--naming=eg.MyNamingStrategy`	选择一个命名策略 (`NamingStrategy`)
`--properties=hibernate.properties`	从文件中读取数据库属性
`--config=hibernate.cfg.xml`	指定一个 `.cfg.xml` 文件

你可以在你的应用程序中嵌入 `SchemaValidator` ：

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

20.1.8. 使用Ant进行schema校验

你可以在Ant脚本中调用 `SchemaValidator`：

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

第 21 章 示例：父子关系(Parent Child Relationships)

目录

刚刚接触Hibernate的人大多是从父子关系（parent / child type relationship）的建模入手的。父子关系的建模有两种方法。由于种种原因，最方便的方法是把 `Parent` 和 `Child` 都建模成实体类，并创建一个从 `Parent` 指向 `Child` 的<one-to-many>关联，对新手来说尤其如此。还有一种方法，就是将 `Child` 声明为一个 <composite-element> （组合元素）。 事实上在Hibernate中one to many关联的默认语义远没有composite element贴近parent / child关系的通常语义。下面我们会阐述如何使用带有级联的双向一对多关联*(bidirectional one to many association with cascades)*去建立有效、优美的parent / child关系。这一点也不难！

21.1. 关于collections需要注意的一点

Hibernate collections被当作其所属实体而不是其包含实体的一个逻辑部分。这非常重要！它主要体现为以下几点：

- 当删除或增加collection中对象的时候，collection所属者的版本值会递增。
- 如果一个从collection中移除的对象是一个值类型(value type)的实例，比如composite element，那么这个对象的持久化状态将会终止，其在数据库中对应的记录会被删除。同样的，向collection增加一个value type的实例将会使之立即被持久化。
- 另一方面，如果从一对多或多对多关联的collection中移除一个实体，在缺省情况下这个对象并不会被删除。这个行为是完全合乎逻辑的－－改变一个实体的内部状态不应该使与它关联的实体消失掉！同样的，向collection增加一个实体不会使之被持久化。

实际上，向Collection增加一个实体的缺省动作只是在两个实体之间创建一个连接而已，同样移除的时候也只是删除连接。这种处理对于所有的情况都是合适的。对于父子关系则是完全不适合的，在这种关系下，子对象的生存绑定于父对象的生存周期。

21.2. 双向的一对多关系(Bidirectional one-to-many)

假设我们要实现一个简单的从Parent到Child的<one-to-many>关联。

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

如果我们运行下面的代码

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate会产生两条SQL语句：

- 一条 `INSERT` 语句，为 `c` 创建一条记录
- 一条 `UPDATE` 语句，创建从 `p` 到 `c` 的连接

这样做不仅效率低，而且违反了列 `parent_id` 非空的限制。我们可以通过在集合类映射上指定 `not-null="true"` 来解决违反非空约束的问题：

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>
```

然而，这并非是推荐的解决方法。

这种现象的根本原因是从 `p` 到 `c` 的连接（外键parent_id）没有被当作 `Child` 对象状态的一部分，因而没有在INSERT语句中被创建。因此解决的办法就是把这个连接添加到Child的映射中。

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

（我们还需要为类 `Child` 添加 `parent` 属性）

现在实体 `Child` 在管理连接的状态，为了使collection不更新连接，我们使用 `inverse` 属性。

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

下面的代码是用来添加一个新的 `Child`

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

现在，只会有一条 `INSERT` 语句被执行！

为了让事情变得井井有条，可以为 `Parent` 加一个 `addChild()` 方法。

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

现在，添加 `Child` 的代码就是这样

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

21.3. 级联生命周期（Cascading lifecycle）

需要显式调用 `save()` 仍然很麻烦，我们可以用级联来解决这个问题。

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

这样上面的代码可以简化为：

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

同样的，保存或删除 `Parent` 对象的时候并不需要遍历其子对象。 下面的代码会删除对象 `p` 及其所有子对象对应的数据库记录。

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

然而，这段代码

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

不会从数据库删除 `c` ；它只会删除与 `p` 之间的连接（并且会导致违反 `NOT NULL` 约束，在这个例子中）。你需要显式调用 `delete()` 来删除 `Child` 。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

在我们的例子中，如果没有父对象，子对象就不应该存在，如果将子对象从collection中移除，实际上我们是想删除它。要实现这种要求，就必须使用 `cascade="all-delete-orphan"` 。

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

注意：即使在collection一方的映射中指定 `inverse="true"` ，级联仍然是通过遍历collection中的元素来处理的。如果你想要通过级联进行子对象的插入、删除、更新操作，就必须把它加到collection中，只调用 `setParent()` 是不够的。

21.4. 级联与 `未保存值` （Cascades and `unsaved-value` ）

假设我们从 `Session` 中装入了一个 `Parent` 对象，用户界面对其进行了修改，然后希望在一个新的Session里面调用 `update()` 来保存这些修改。对象 `Parent` 包含了子对象的集合，由于打开了级联更新，Hibernate需要知道哪些Child对象是新实例化的，哪些代表数据库中已经存在的记录。我们假设 `Parent` 和 `Child` 对象的标识属性都是自动生成的，类型为 `java.lang.Long` 。Hibernate会使用标识属性的值，和version 或 timestamp 属性，来判断哪些子对象是新的。(参见第 10.7 节 “自动状态检测”.) 在 *Hibernate3* 中,显式指定 *`unsaved-value`* 不再是必须的了。

下面的代码会更新 `parent` 和 `child` 对象，并且插入 `newChild` 对象。

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Well, that's all very well for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since Hibernate can't use the identifier property to distinguish between a newly instantiated object (with an identifier assigned by the user) and an object loaded in a previous session. In this case, Hibernate will either use the timestamp or version property, or will actually query the second-level cache or, worst case, the database, to see if the row exists.

这对于自动生成标识的情况是非常好的，但是自分配的标识和复合标识怎么办呢？这是有点麻烦，因为Hibernate没有办法区分新实例化的对象（标识被用户指定了）和前一个Session装入的对象。在这种情况下，Hibernate会使用timestamp或version属性，或者查询第二级缓存，或者最坏的情况，查询数据库，来确认是否此行存在。

21.5. 结论

这里有不少东西需要融会贯通，可能会让新手感到迷惑。但是在实践中它们都工作地非常好。大部分Hibernate应用程序都会经常用到父子对象模式。

在第一段中我们曾经提到另一个方案。上面的这些问题都不会出现在 `<composite-element>` 映射中，它准确地表达了父子关系的语义。很不幸复合元素还有两个重大限制:复合元素不能拥有collections，并且，除了用于惟一的父对象外，它们不能再作为其它任何实体的子对象。

第 22 章 示例：Weblog 应用程序

目录

22.1. 持久化类

下面的持久化类表示一个weblog和在其中张贴的一个贴子。他们是标准的父/子关系模型，但是我们会用一个有序包（ordered bag)而非集合(set)。

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

22.2. Hibernate 映射

下列的XML映射应该是很直白的。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd

<hibernate-mapping package="eg">
    <class
        name="Blog"
        table="BLOGS" >

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

22.3. Hibernate 代码

下面的类演示了我们可以使用Hibernate对这些类进行的一些操作。

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
```

```
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, Strir
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
```

```
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class,
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }
```

```
    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
```

```
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateExceptior

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

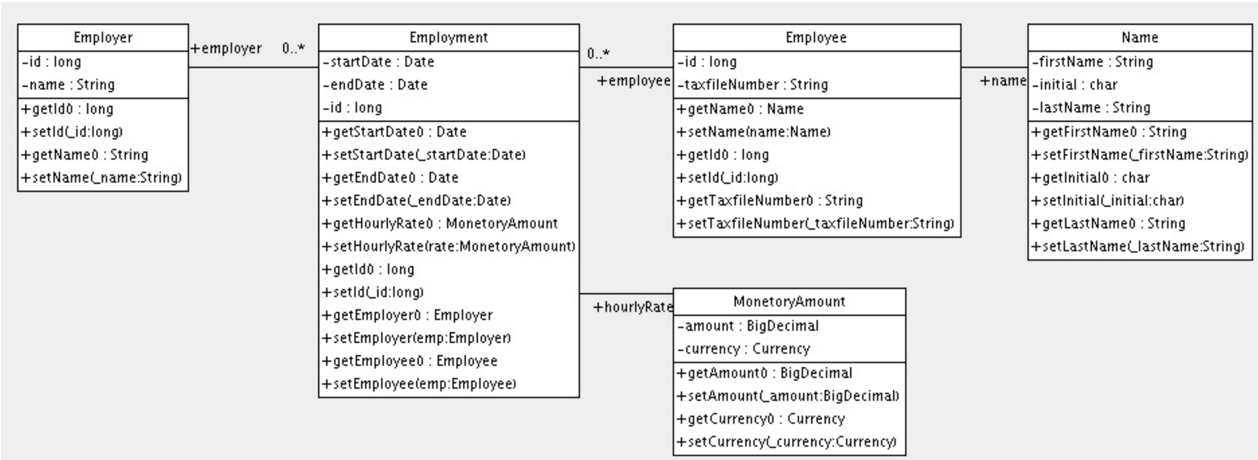
第 23 章 示例：复杂映射实例

目录

本章展示了一些较为复杂的关系映射。

23.1. Employer（雇主)/Employee(雇员)

下面关于 `Employer` 和 `Employee` 的关系模型使用了一个真实的实体类(`Employment`)来表述，这是因为对于相同的雇员和雇主可能会有多个雇佣时间段。 对于金额和雇员姓名，用Components建模。



映射文件可能是这样:

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null=
        <many-to-one name="employee" column="employee_id" not-null=

    </class>

    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

用 `SchemaExport` 生成表结构。

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

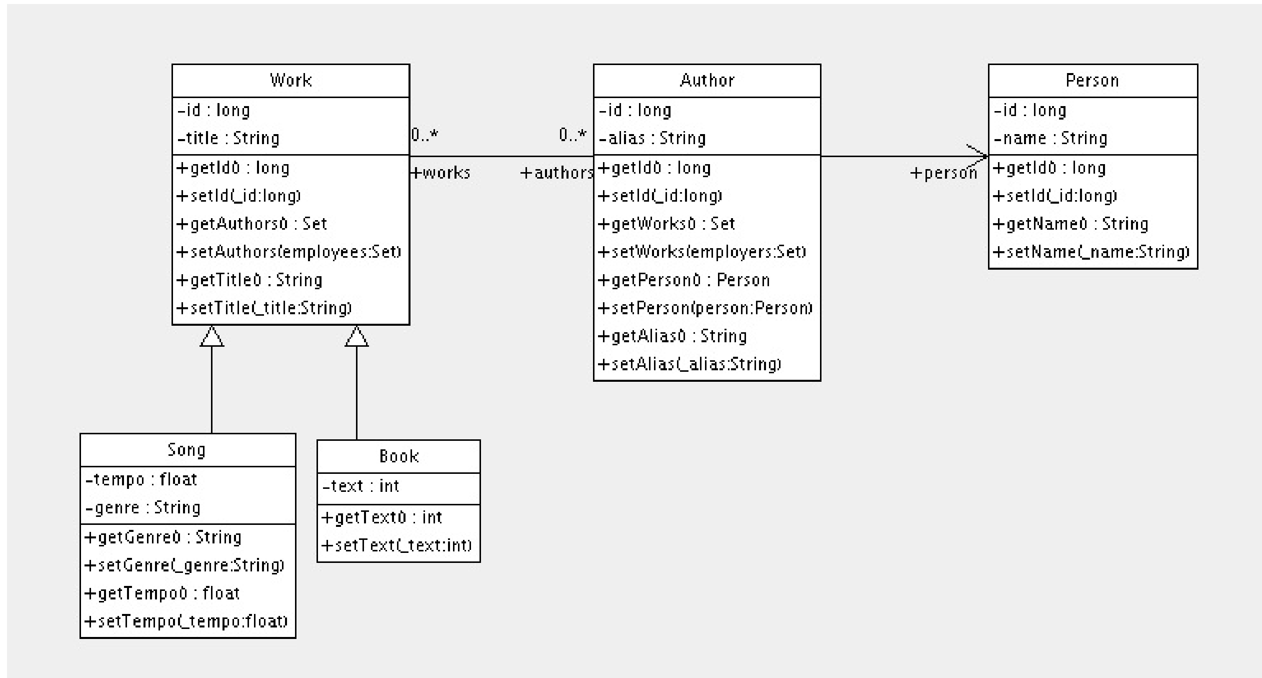
create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id)
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id)
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

23.2. Author(作家)/Work(作品)

考虑下面的 `Work` , `Author` 和 `Person` 模型的关系。 我们用多对多关系来描述 `Work` 和 `Author` ， 用一对一关系来描述 `Author` 和 `Person` ， 另一种可能性是 `Author` 继承 `Person` 。



下面的映射文件正确的描述了这些关系：

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
```

```
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Pe
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

映射中有4个表。 `works` , `authors` 和 `persons` 分别保存着work，author和person的数据。 `author_work` 是authors和works的关联表。 表结构是由 `SchemaExport` 生成的。

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

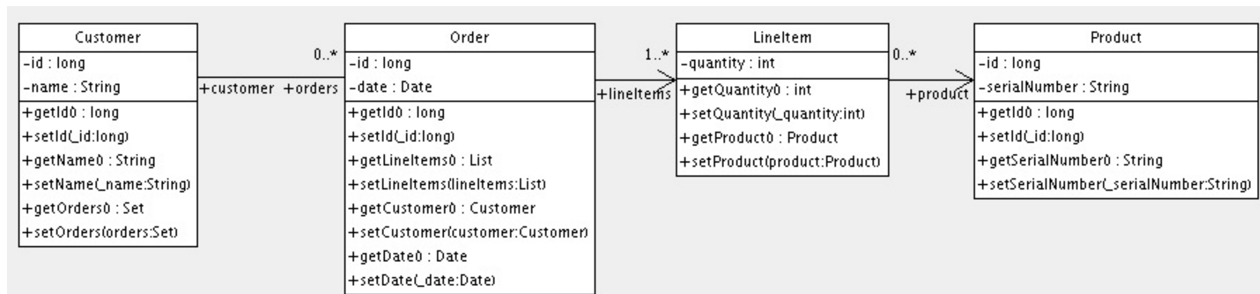
create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) reference
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references
```

23.3. Customer(客户)/Order(订单)/Product(产品)

现在来考虑 `Customer`，`Order`，`LineItem` 和 `Product` 关系的模型。`Customer` 和 `Order` 之间 是一对多的关系，但是我们怎么来描述 `Order` / `LineItem` / `Product` 呢？ 我可以把 `LineItem` 作为描述 `Order` 和 `Product` 多对多关系的关联类，在Hibernate，这叫做组合元素。



映射文件如下：

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

`customers`, `orders`, `line_items` 和 `products` 分别保存着customer, order, order line item 和 product的数据。 `line_items` 也作为连接orders 和 products的关联表。

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references c
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) reference
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references
```

23.4. 杂例

这些例子全部来自于Hibernate的test suite，同时你也可以找到其他有用的例子。可以参考Hibernate的 `test` 目录。

TODO: put words around this stuff

23.4.1. "Typed" one-to-one association

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class>
```

23.4.2. Composite key example

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
```

```
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
```

```
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class>
```

23.4.3. 共有组合键属性的多对多(Many-to-many with shared composite key attribute)

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

23.4.4. Content based discrimination

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>

    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class>
```

23.4.5. Associations on alternate keys

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class>
```

第 24 章 最佳实践(Best Practices)

设计细颗粒度的持久类并且使用 <component> 来实现映射。

使用一个 `Address` 持久类来封装 `street` , `suburb` , `state` , `postcode` . 这将有利于代码重用和简化代码重构(refactoring)的工作。

对持久类声明标识符属性(identifier properties)。

Hibernate中标识符属性是可选的，不过有很多原因来说明你应该使用标识符属性。我们建议标识符应该是“人造”的(自动生成，不涉及业务含义)。

使用自然键(natural keys)标识

对所有的实体都标识出自然键，用 <natural-id> 进行映射。实现 `equals()` 和 `hashCode()` ，在其中用组成自然键的属性进行比较。

为每个持久类写一个映射文件

不要把所有的持久类映射都写到一个大文件中。把 `com.eg.Foo` 映射到 `com/eg/Foo.hbm.xml` 中， 在团队开发环境中， 这一点显得特别有意义。

把映射文件作为资源加载

把映射文件和他们的映射类放在一起进行部署。

考虑把查询字符串放在程序外面

如果你的查询中调用了非ANSI标准的SQL函数，那么这条实践经验对你适用。把查询字符串放在映射文件中可以让程序具有更好的可移植性。

使用绑定变量

就像在JDBC编程中一样， 应该总是用占位符"?"来替换非常量值，不要在查询中用字符串值来构造非常量值！更好的办法是在查询中使用命名参数。

不要自己来管理JDBC connections

Hibernate允许应用程序自己来管理JDBC connections，但是应该作为最后没有办法的办法。如果你不能使用Hibernate内建的connections providers，那么考虑实现自己来实现 `org.hibernate.connection.ConnectionProvider`

考虑使用用户自定义类型(custom type)

假设你有一个Java类型，来自某些类库，需要被持久化，但是该类没有提供映射操作需要的存取方法。那么你应该考虑实现 `org.hibernate.UserType` 接口。这种办法使程序代码写起来更加自如，不再需要考虑类与Hibernate type之间的相互转换。

在性能瓶颈的地方使用硬编码的JDBC

In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. But please, wait until you *know* something is a bottleneck. And don't assume that direct JDBC is necessarily faster. If you need to use direct JDBC, it might be worth opening a Hibernate `Session` and using that JDBC connection. That way you can still use the same transaction strategy and underlying connection provider. 在系统中对性能要求很严格的一些部分，某些操作也许直接使用JDBC会更好。但是请先确认这的确是一个瓶颈，并且不要想当然认为JDBC一定会更快。如果确实需要直接使用JDBC，那么最好打开一个 Hibernate `Session` 然后从 `Session` 获得connection，按照这种办法你仍然可以使用同样的transaction策略和底层的connection provider。

理解 `Session` 清洗（ flushing）

Session会不时的向数据库同步持久化状态，如果这种操作进行的过于频繁，性能会受到一定的影响。有时候你可以通过禁止自动flushing，尽量最小化非必要的flushing操作，或者更进一步，在一个特定的transaction中改变查询和其它操作的顺序。

在三层结构中，考虑使用托管对象（detached object）

当使用一个servlet / session bean 类型的架构的时候, 你可以把已加载的持久对象在session bean层和servlet / JSP 层之间来回传递。使用新的session来为每个请求服务，使用 `Session.merge()` 或者 `Session.saveOrUpdate()` 来与数据库同步。

在两层结构中，考虑使用长持久上下文(long persistence contexts).

为了得到最佳的可伸缩性，数据库事务(Database Transaction)应该尽可能的短。但是，程序常常需要实现长时间运行的“应用程序事务*(Application Transaction)”*，包含一个从用户的观点来看的原子操作。这个应用程序事务可能跨越多次从用户请求到得到反馈的循环。用脱管对象(与session脱离的对象)来实现应用程序事务是常见的。或者，尤其在两层结构中，把Hibernate Session从JDBC连接中脱离开，下次需要用的时候再连接上。绝不要把一个Session用在多个应用程序事务(Application Transaction)中，否则你的数据可能会过期失效。

不要把异常看成可恢复的

这一点甚至比“最佳实践”还要重要，这是“必备常识”。当异常发生的时候，必须要回滚 `Transaction` ，关闭 `Session` 。如果你不这样做的话，Hibernate无法保证内存状态精确的反应持久状态。尤其不要使用 `Session.load()` 来判断一个给定标识符的对象实例在数据库中是否存在，应该使用 `Session.get()` 或者进行一次查询.

对于关联优先考虑lazy fetching

谨慎的使用主动抓取(eager fetching)。对于关联来说，若其目标是无法在第二级缓存中完全缓存所有实例的类，应该使用代理(proxies)与/或具有延迟加载属性的集合(lazy collections)。若目标是可以被缓存的，尤其是缓存的命中率非常高的情况下，应该使用 `lazy="false"` ，明确的禁止掉eager fetching。如果那些特殊的确实适合使用join fetch 的场合，请在查询中使用 `left join fetch` 。

使用*open session in view*模式，或者执行严格的装配期*(assembly phase)*策略来避免再次抓取数据带来的问题

Hibernate让开发者们摆脱了繁琐的*Data Transfer Objects* (DTO)。在传统的EJB结构中，DTO有双重作用：首先，他们解决了entity bean无法序列化的问题；其次，他们隐含地定义了一个装配期，在此期间，所有在view层需要用到的数据，都被抓取、集中到了DTO中，然后控制才被装到表示层。Hibernate终结了第一个作用。然而，除非你做好了在整个渲染过程中都维护一个打开的持久化上下文(session)的准备，你仍然需要一个装配期（想象一下，你的业务方法与你的表示层有严格的契约，数据总是被放置到托管对象中）。这并非是Hibernate的限制！这是实现安全的事务化数据访问的基本需求。

考虑把Hibernate代码从业务逻辑代码中抽象出来

把Hibernate的数据存取代码隐藏到接口(interface)的后面，组合使用*DAO*和*Thread Local Session*模式。通过Hibernate的 `UserType` ，你甚至可以用硬编码的JDBC来持久化那些本该被Hibernate持久化的类。 (该建议更适用于规模足够大应用软件中，对于那些只有5张表的应用程序并不适合。)

不要用怪异的连接映射

多对多连接用得好的例子实际上相当少见。大多数时候你在“连接表”中需要保存额外的信息。这种情况下，用两个指向中介类的一对多的连接比较好。实际上，我们认为绝大多数的连接是一对多和多对一的，你应该谨慎使用其它连接风格，用之前问自己一句，是否真的必须这么做。

偏爱双向关联

单向关联更加难于查询。在大型应用中，几乎所有的关联必须在查询中可以双向导航。